Optimized Software Synthesis for Digital Signal Processing Algorithms: An Evolutionary Approach

J. Teich, E. Zitzler
Institute TIK
Swiss Federal Institute of Technology
Gloriastr. 35, CH-8092 Zurich
Switzerland

S. S. Bhattacharyya
EE Dept. and UMIACS
University of Maryland
College Park MD 20742
U.S.A.

Abstract - Based on the model of synchronous data flow (SDF) [13], so called single appearance schedules are known to provide memory-optimal schedules. Among these, the problem of buffer memory optimization is treated: (1) An Evolutionary Algorithm (EA) is applied to efficiently explore the (in general) exponential search space of actor firing orders. (2) For each order, the buffer costs are evaluated by applying a dynamic programming post-optimization step (GDPPO).

Introduction

Dataflow specifications are widespread in areas of digital signal and image processing. Synchronous dataflow (SDF) graphs [13] present a class of dataflow in which the nodes, called *actors* have a simple firing rule: The number of data values (*tokens, samples*) produced and consumed by each actor is fixed and known at compile-time. The SDF model is used in many industrial DSP design tools, e.g., SPW by Cadence, COSSAP by Synopsys, as well as in research-oriented environments, e.g., [4, 12, 15]. In general, a code generation method that generates inline code from a given actor schedule (sequence of actor firings) is assumed. With this model, so called *single appearance schedules*, where each actor appears only once in a schedule, are evidently program memory optimal. Results on the existence of such schedules have already been published for general SDF graphs [2].

In this paper, we treat the problem of generating single appearance schedules that minimize the amount of required buffer memory for the class of acyclic SDF graphs. Such a methodology may be considered as part of a general framework that considers general SDF graphs and generates schedules for acyclic subgraphs using our approach [3].

Motivation

Given is an acyclic SDF graph in the following. The number of single appearance schedules that must be investigated is at least equal to the number of topological sorts of actors in the graph. This number is not polynomially bounded; e.g., a complete bipartite graph with $2n$ nodes has $(n!)^2$ possible topological sorts. This complexity prevents techniques based on enumeration from being applied sucessfully. In [3], a heuristic called APGAN (for algorithm for pairwise grouping of adjacent nodes (acyclic version)) has been

developed that constructs a schedule with the objective to minimize buffer memory. This procedure has been shown to give optimal results for a certain class of graphs having a regular structure. Also, a complementary procedure called RPMC (for recursive partitioning by minimum cuts) has been proposed that works well on more irregular (e.g., randomly generated) graph structures. Although being computationally efficient, these heuristics sometimes produce results that are far from optimal, see Example 1.

Example 1 *We consider two testgraphs and compare different buffer optimization algorithms (see Table 1). For the simple graph in Fig. 1b), already 362 880 different topological sorts (actor firing orders) may be constructed with buffer requirements ranging between 3003 and 15 705 memory units. The 2nd graph is randomly generated with 50 nodes. The 1st method in Table 1 uses an Evolutionary Algorithm (EA) that performs 3000 fitness calculations, the 2nd is the APGAN heuristic, the 3rd is a* Monte Carlo *simulation (3000 random tries), and the 4th an exhaustive search procedure which did not terminate in the second case.*

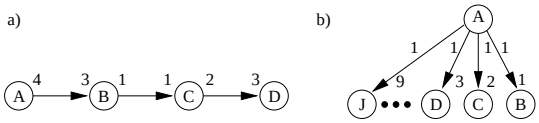

Figure 1: Simple SDF graphs.

	Graph 1		Graph 2	
method	best cost	runtime (s)	best cost	runtime (s)
EA	3003	4.57	669 380	527.87
APGAN	3015	0.02	15 063 956	1.88
RPMC	3151	0.03	1 378 112	2.03
Monte Carlo	3014	3.3	2 600 349	340.66
Exh. Search	3003	373	?	?

Table 1: Analysis of existing heuristics on simple testgraphs. The run-times were measured on a SUN SPARC 20.

The motivation of the following work was to develop a methodology that is *cost-competitive* against existing approaches such as APGAN and RPMC, and at the same time *run-time tolerable*.

Proposed Approach

Here, we use a unique two-step approach to find buffer-minimal schedules:
(1) An Evolutionary Algorithm (EA) is used to efficiently explore the space of topological sorts of actors given an SDF graph using a population of N individuals each of which encodes a topological sort.
(2) For each topological sort, a buffer optimal schedule is constructed based on a well-known dynamic programming post optimization step [3] that determines a loop nest by parenthesization (see Fig. 2) that is buffer cost optimal

(for the given topological order of actors). The run-time of this optimization step is $\mathcal{O}(N^3)$.

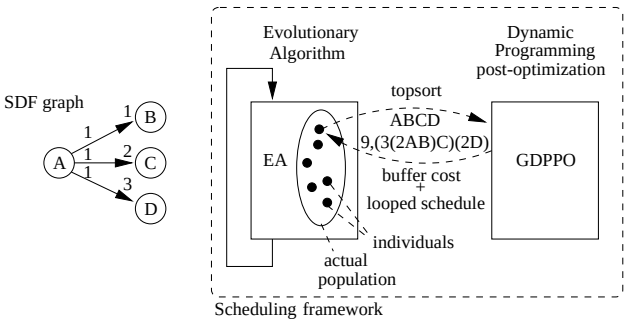

Figure 2: Overview of the scheduling framework using Evolutionary Algorithms and Dynamic Programming (GDPPO: generalized dynamic programming post optimization for optimally parenthesizing actor orderings [3]) for constructing buffer memory optimal schedules.

The overall picture of the scheduling framework is depicted in Fig. 2. Details on the optimization procedure and the cost function will be explained in the following. The total run-time of the algorithm is $\mathcal{O}(Z\,N^3)$ where Z is the number of evocations of the dynamic program post-optimizer.

An Evolutionary Approach for Memory Optimization

The SDF-scheduling framework

Definition 1 (SDF graph) *An SDF graph [13] G denotes a 5-tuple $G = (V, A, produced, consumed, delay)$ where*

- *V is the set of nodes (*actors*) ($V = \{v_1, v_2, \cdots, v_{\mathrm{K}}\}$).*
- *A is the set of directed arcs. With $source(\alpha)$ ($sink(\alpha)$), we denote the source node (target node) of an arc $\alpha \in A$.*
- *$produced : A \rightarrow \mathbf{N}$ denotes a function that assigns to each directed arc $\alpha \in A$ the number of produced tokens $produced(\alpha)$ per invocation of actor $source(\alpha)$.*
- *$consumed : A \rightarrow \mathbf{N}$ denotes a function that assigns to each directed arc $\alpha \in A$ the number of consumed tokens per invocation of actor $sink(\alpha)$.*
- *$delay : A \rightarrow \mathbf{N}_0$ denotes the function that assigns to each arc $\alpha \in A$ the number of initial tokens $delay(\alpha)$.*

A *schedule* is a sequence of actor firings. A properly-constructed SDF graph is compiled by first constructing a finite schedule S that fires each actor at least once, does not deadlock, and produces no net change in the number of tokens on queues associated with each arc. When such a schedule is repeated infinitely, we call the resulting infinite sequence of actor firings a *valid*

periodic schedule, or simply *valid schedule*. Graphs with this property are called *consistent*. For such a graph, the minimum number of times each actor must execute may be computed efficiently [13] and captured by a function $q : V \rightarrow \mathbf{N}$.

Example 2 *Figure 1a) shows an SDF graph with nodes labeled A, B, C, D, respectively. The minimal number of actor firings is obtained as $q(A) = 9$, $q(B) = q(C) = 12$, $q(D) = 8$. The schedule $(\infty(2ABC)DABCDBC(2ABCD)A(2BC)(2ABC)A(2BCD))$ represents a valid schedule. A parenthesized term $(n\ S_1\ S_2\ \cdots\ S_k)$ specifies n sucessive firings of the "subschedule" $S_1\ S_2\ \cdots\ S_k$.*

Each parenthesized term $(n\ S_1\ S_2\ \cdots\ S_k)$ is referred to as *schedule loop* having *iteration count* n and *iterands* $S_1, S_2, \cdots, S_k$. We say that a schedule for an SDF graph is a *looped schedule* if it contains zero or more schedule loops. A schedule is called *single appearance schedule* if it contains only one appearance of each actor. In general, a schedule of the form $(\infty\ (q(N_1)N_1)\ (q(N_2)N_2)\ \cdots\ (q(N_{\mathrm{K}})N_{\mathrm{K}}))$ where N_i denotes the (label of the) ith node of a given SDF graph, and K denotes the number of nodes of the given graph, is called *flat single appearance schedule*.

Code generation and buffer cost model

Given an SDF graph, we consider code generation by inlining an actor code block for each actor appearance in the schedule. The resulting sequence of code blocks is encapsulated within an infinite loop to generate a software implementation. Each schedule loop thereby is translated into a loop in the target code. In order to determine the amount of data needed to store the tokens that accumulate on each arc during the evolution of a schedule S, we define the cost function $buffer\_memory(S) = \sum_{\alpha \in A} max\_tokens(\alpha, S)$, where $max\_tokens(\alpha, S)$ denotes the maximum number of tokens that accumulate on arc α during the execution of schedule S.[1]

Example 3 *Consider the flat schedule $(\infty(9A)(12B)(12C)(8D))$ for the SDF graph shown in Fig. 1a). This schedule has a buffer memory requirement of $36 + 12 + 24 = 72$. Similarly, the buffer memory requirement of the schedule $(\infty(3(3A)(4B))(4(3C)(2D)))$ is $12 + 12 + 6 = 30$.*

Related Work

The interaction between instruction scheduling and register allocation in procedural language compilers has been studied extensively [10, 1], and optimal management of this interaction is intractable [9]. More recently, the issue of optimal storage allocation has been examined in the context of high-level synthesis for iterative DSP programs [6], and code generation for embedded processors that have highly irregular instruction formats and register sets

[1] Note that this model of buffering – maintaining a separate memory buffer for each data flow edge – is convenient and natural for code generation. More technical advantages of this model are elaborated in [3].

[14, 11]. However, because of their focus on fine-grain scheduling, the above efforts apply to a homogeneous data flow model – that is, a model in which each computation (node) produces and consumes a single value to/from each incident edge. Similarly, Fabri [7] and others have examined the problem of managing pools of logical buffers that have varying sizes, given a set of buffer lifetimes, but such efforts are also in isolation of the scheduling problems that we face in the context of general SDF graphs.

Why Use an Evolutionary Algorithm?

From Example 1, it became clear that there exist simple graphs for which there is a big gap between the quality of solution obtained using heuristics such as APGAN and an Evolutionary Algorithm (EA). If the run-time of such an iterative approach is still affordable, a performance gap of several orders of magnitude may be avoided.

Exploration of topological sorts using the EA

Given an acyclic SDF graph, one major difficulty consists in finding a coding of feasible topological sorts. Details on the coding scheme are given in the next section that deals with all implementation issues of the evolutionary search procedure.

Dynamic programming post optimization

In [3], it has been shown that given a topological sort of actors of a consistent, delayless and acyclic SDF graph G, a single-appearance schedule can be computed that minimizes buffer memory over all single-appearance schedules for G that have the given lexical ordering. Such a minimum buffer memory schedule can be computed using a dynamic programming technique called GDPPO.

Example 4 *Consider again the SDF graph in Fig. 1a). With* $q(A) = 9$, $q(B) = q(C) = 12$, *and* $q(D) = 8$, *an optimal schedule is* $(\infty(3(3A)(4B))(4(3C)(2D)))$ *with a buffer cost of 30. Given the topological order of nodes* A, B, C, D *as imposed by the arcs of* G, *this schedule is obtained by parenthesization of the string. Note that this optimal schedule contains a break in the chain at some actor* k, $1 \leq k \leq \mathrm{K} - 1$. *Because the parenthesization is optimal, the chains to the left of* k *and to the right of* k *must also be parenthesized optimally. This structure of the optimization problem is essential for dynamic programming.*

Parametrization of the Evolutionary Algorithm

The EA, which is based on a generational model, works on a set (population) of topological sorts (encoded in the individuals) by means of selection, crossover, and mutation. Thereby, the fitness of an individual is equal to the buffer memory cost induced by the corresponding topological sort—the buffer memory requirements are calculated by the GDPPO procedure.

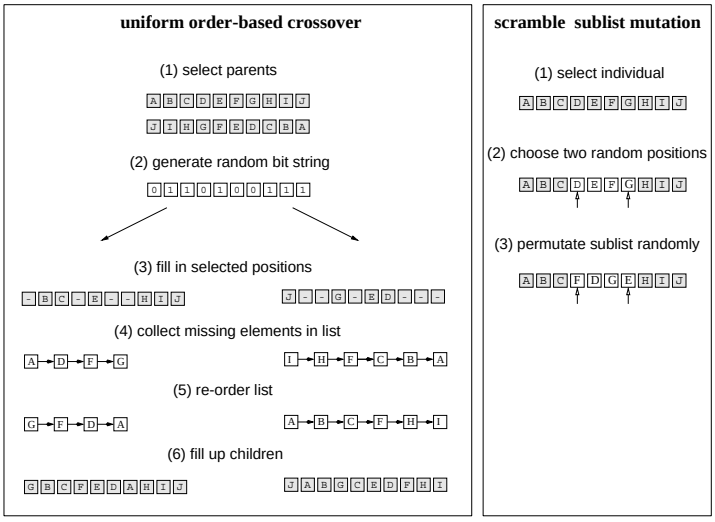

Figure 3: Order-based crossover and mutation. The exemplary individuals refer to the SDF graph depicted in Figure 1b).

Coding and Repair Mechanism

Our combinatorial optimization problem naturally suggests to use an order-based representation where each individual encodes a permutation over the set of nodes. A simple repair mechanism, transforming permutations into topological sorts, guarantees that every genotype can be mapped to a valid topological sort. Thus, there are no infeasible individuals in the population. On the other hand, since each topological sort is simultaneously a permutation, the whole search space is covered by this representation.

In each step of the repair algorithm, a node with an indegree equal to zero is chosen and removed from the graph (together with the incident edges). The order in which the nodes appear determines the topological sort. The tie between several nodes with no ingoing edges is normally broken by random. Our algorithm, however, always selects the node at the leftmost position within the permutation. This ensures on the one hand that each individual is mapped unambiguously to one topological sort, and on the other hand that every topological sort has at least one encoding.

Genetic Operators

The selection scheme chosen is *tournament selection* where a fixed number of individuals is picked out randomly, and the individual having the best fitness value (lowest buffer cost) within this group is copied to the new population; this process is repeated until the new population has been filled up. Additionally, an *elitist strategy* has been implemented: the best individual per generation is preserved by simply copying it to the population of the next generation.

Since individuals encode permutations, we applied a crossover operator,

named uniform order-based crossover [5][8], which preserves the permutation property. How it works is shown in Figure 3 on the left side.

Mutation is done by permuting the elements between two selected positions, whereas both the positions and the subpermutation are chosen by random. That is what Davis calls *scramble sublist mutation* [5]. An example which shows the mutation mechanism is given in Figure 3 on the right side.

Crossover Probability and Mutation Probability

To the recombination operator as well as to the mutation operator, probabilities for their application are associated, namely the crossover probability p_c and the mutation probability p_m. We investigated several different p_c-p_m-combinations on a few random graphs containing 50 nodes.[2]

We have chosen a population size of 30 individuals. The crossover rates we tested are 0, 0.2, 0.4, 0.6, and 0.8, while the mutation rates cover the range from 0 to 0.4 by a step size of 0.1. Altogether, the EA ran with 24 various p_c-p_m-settings on every test graph. It stopped after 3000 fitness evaluations. For each combination we took the average fitness (buffer cost) over ten independent runs.

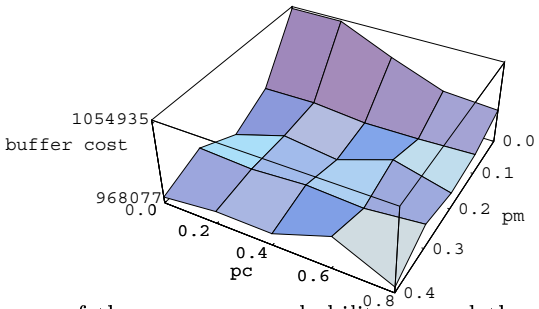

Figure 4: Influence of the crossover probability p_c and the mutation probability p_m on the average fitness for a particular test graph (3000 fitness evaluations).

The results for a particular graph are visualized in Figure 4; the results for the other random test graphs look similar. Obviously, mutation is essential to this problem. Setting p_m to 0 leads to the worst results of all probabilty combinations. If p_m is greater than 0, the obtained average buffer costs are significantly smaller—almost independently of the choice of p_c. As can be seen in Figure 5 this is due to premature convergence. The curve representing the performance for $p_c = 0.2$ and $p_m = 0$ goes horizontally after about 100 fitness evaluations. No new points in the search space are explored.

On the other hand, the impact of the crossover operator on the overall performance is not as great as that of the mutation operator. With no mutation at all, increasing p_c yields decreased average buffer cost. But this is not the same to cases where $p_m > 0$. The curve for $p_c = 0.6$ and $p_m = 0.2$ in Figure 5 bears out this observation. Beyond it, for this particular test graph

[2] Graphs consisting of less nodes are not very well suited to obtain reliable values for p_c and p_m, because the optimum is yet reached after a few generations, in most cases.

a mutation probability of $p_m = 0.2$ and a crossover probability of $p_c = 0$ leads to best performance. Nevertheless, with respect to the results on other test graphs, we found a crossover rate of $p_c = 0.2$ and a mutation rate of $p_m = 0.4$ to be most appropriate for this problem.

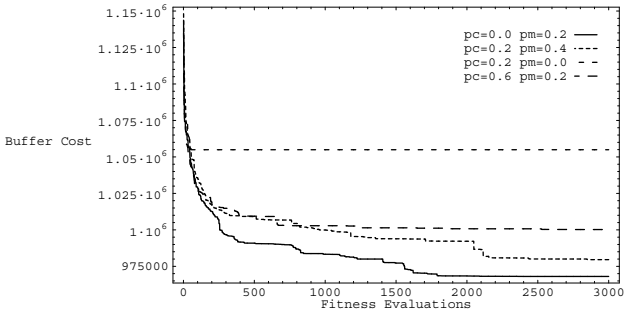

Figure 5: Performance of the Evolutionary Algorithm according to four different p_c-p_m-combinations; each graph represents the average of ten runs.

Experiments

To evaluate the performance of the EA we tested it on several practical examples of acyclic, multirate SDF graphs as well as on 200 acyclic random graphs, each containing 50 nodes and having 100 edges in average. The obtained results were compaired against the outcomes produced by APGAN, RPMC, a random search strategy known as Monte Carlo (MC), and a Hill Climbing approach (HC). We also tried a modified version of the EA which first runs APGAN and then inserts the computed topological sort into the initial population.

Table 2 shows the results of applying GDPPO to the schedules generated by the various heuristics on several practical SDF graphs; the satellite receiver example is taken from [16], whereas the other examples are the same as considered in [3]. The probabilistic algorithms ran once on each graph and were aborted after 3000 fitness evaluations. Additionally, an exhaustive search with a maximum run-time of 1 hour was carried out; as it only completed in two cases[3], the search spaces of these problems seem to be rather complex.

In all of the practical benchmark examples in Table 2 the results achieved by the EA equal or surpass those generated by RPMC. Compared to APGAN on these practical examples, the EA is neither inferior nor superior; it shows both better and worse performance in two cases each. Furthermore, the performance of the Hill Climbing approach is almost identical to performance of the EA. The Monte Carlo simulation, however, performs slightly worse than the other probabilistic approaches.

[3] Laplacian pyramid (minimal buffer cost: 99); QMF filterbank, one-sided tree (minimal buffer cost: 108).

[4] The following systems have been considered: 1) fractional decimation; 2) Laplacian pyramid; 3) nonuniform filterbank (1/3, 2/3 splits, 4 channels); 4) nuniform filterbank (1/3, 2/3 splits, 6 channels); 5) QMF nonuniform-tree filterbank; 6) QMF filterbank (one-sided tree); 7) QMF analysis only; 8) QMF tree filterbank (4 channels); 9) QMF tree filterbank (8 channels); 10) QMF tree filterbank (16 channels); 11) satellite receiver.

System	BMLB	APGAN	RPMC	MC	HC	EA	EA + APGAN
1	47	47	52	47	47	47	47
2	95	99	99	99	99	99	99
3	85	137	128	143	126	126	126
4	224	756	589	807	570	570	570
5	154	160	171	165	160	160	159
6	102	108	110	110	108	108	108
7	35	35	35	35	35	35	35
8	46	46	55	46	47	46	46
9	78	78	87	78	80	80	78
10	166	166	200	188	190	197	166
11	1540	1542	2480	1542	1542	1542	1542

Table 2: Comparison of performance on practical examples; the probabilistic algorithms stopped after 3000 fitness evaluations. BMLB stands for a lower buffer limit: buffer memory lower bound.[4]

<	APGAN	RPMC	MC	HC	EA	EA + APGAN
APGAN	0%	34.5%	15%	0%	1%	0%
RPMC	65.5%	0%	29.5%	3.5%	4.5%	2.5%
MC	85%	70.5%	0%	0.5%	0.5%	1%
HC	100%	96.5%	99.5%	0%	70%	57%
EA	99%	95.5%	99.5%	22%	0%	39%
EA + APGAN	100%	97.5%	99%	32.5%	53.5%	0%

Table 3: Comparison of performance on 200 50-actor SDF graphs (3000 fitness evaluations); for each row the numbers represent the fraction of random graphs on which the correspondig heuristic outperforms the other approaches.

Although the results are nearly the same when considering only 1500 fitness evaluations, the EA (as well as Monte Carlo and Hill Climbing) cannot compete with APGAN or RPMC concerning run-time performance. E.g., APGAN needs less than 2.3 second for all graphs on a SUN SPARC 20, while the run-time of the EA varies from 0.1 seconds up to 5 minutes (3000 fitness evaluations).

The results concerning the random graphs are summarized in Table 3; again, the stochastic approaches were aborted after 3000 fitness evaluations.[5] Interestingly, for these graphs APGAN is better only in 15% of all cases than Monte Carlo and only on in two cases better than the EA. On the other hand, it is outperformed by the EA 99% of the time.[6] This is almost identical to the comparison between Hill Climbing and APGAN. As RPMC is known to be better suited for irregular graphs than APGAN [3], its better performance (65.5%) is not surprising when directly compared to APGAN. Although, it is beaten by the EA as well as Hill Climbing in 95.5% and 96.5% of the time, respectively.

In summary it may be said that the EA is superior to both APGAN and RMPC on random graphs. In average the buffer costs achieved by the EA are half the costs computed by APGAN and only a fraction of 63% of the

[5] The EA ran about 9 minutes on each graph, the time for running APGAN was constantly less than 3 seconds.

[6] Considering 1500 fitness calculations, this percentage decreases only minimally to 97.5%.

RPMC outcomes. Moreover, an improvement by a factor 28 can be observed on a particular random graph with respect to APGAN (factor 10 regarding RPMC). Compared to Monte Carlo, it is the same, although the margin is smaller (in average the results of the EA are a fraction of 0.84% of the costs achieved by the Monte Carlo simulation). Hill Climbing, however, might be an alternative to the evolutionary approach; the results shown in Table 3 might suggest a superiority of Hill Climbing, but regarding the absolute buffer costs this hypothesis could not be confirmed (the costs achieved by the EA deviate from the costs produced by Hill Climbing by a factor of 0.19% in average).

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Principles of Compiler Design*. Addison-Wesley, Reading, MA, 1986.

[2] S. Bhattacharyya. Compiling data flow programs for digital signal processing. Technical Report UCB/ERL M94/52, Electronics Research Laboratory, UC Berkeley, July 1994.

[3] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell, MA, 1996.

[4] J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal on Computer Simulation*, 4:155–182, 1991.

[5] Lawrence Davis. *Handbook of Genetic Algorithms*, chapter 6, pages 72–90. Van Nostrand Reinhold, New York, 1991.

[6] T. C. Denk and K. K. Parhi. Lower bounds on memory requirements for statically scheduled dsp programs. *J. of VLSI Signal Processing*, pages 247–264, 1996.

[7] J. Fabri. *Automatic Storage Optimization*. UMI Research Press, 1982.

[8] B. R. Fox and M. B. McMahon. Genetic operators for sequencing problems. In Gregory J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 284–300. Morgan Kaufmann, San Mateo, California, 1991.

[9] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, 1979.

[10] W.-C. Hsu. Register allocation and code scheduling for load/store architectures. Technical report, Department of Computer Science, University of Wisconsin at Madison, 1987.

[11] D. J. Kolson, A. N. Nicolau, N. Dutt, and K. Kennedy. Optimal register assignment to loops for embedded code generation. *ACM Trans. on Design Automation of Electronic Systems*, 1(2):251–279, 1996.

[12] R. Lauwereins, M. Engels, J. A. Peperstraete, E. Steegmans, and J. Van Ginderdeuren. Grape: A CASE tool for digital signal parallel processing. *IEEE ASSP Magazine*, 7(2):32–43, April 1990.

[13] E.A. Lee and D.G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[14] P. Marwedel and G. Goossens (eds.). *Code generation for embedded processors*. Kluwer Academic Publishers, Norwell, MA, 1995.

[15] S. Ritz, M. Pankert, and H. Meyr. High level software synthesis for signal processing systems. In *Proc. Int. Conf. on Application-Specific Array Processors*, pages 679–693, Berkeley, CA, 1992.

[16] S. Ritz, M. Willems, and H. Meyr. Scheduling for optimum data memory compaction in block diagram oriented software synthesis. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, volume 4, pages 2651–2654, May 1995.