Parallel Problem Solving from Nature – PPSN V, Amsterdam, The Netherlands, September 1998, pages 885–894. © Springer.

Buffer memory optimization in DSP applications An Evolutionary Approach

Jürgen Teich and Eckart Zitzler[1] and Shuvra Bhattacharyya[2]

[1] Institute TIK, Swiss Federal Institute of Technology, CH-8092 Zurich, Switzerland
[2] EE Dept. and UMIACS, University of Maryland, College Park MD 20742, U.S.A.

Abstract. In the context of digital signal processing, synchronous data flow (SDF) graphs [12] are widely used for specification. For these, so called single appearance schedules provide program memory-optimal uniprocessor implementations. Here, buffer memory minimized schedules are explored among these using an *Evolutionary Algorithm* (EA). Whereas for a restricted class of graphs, there exist optimal polynomial algorithms, these are not exact and may provide poor results when applied to arbitrary, i.e., randomly generated graphs. We show that a careful EA implementation may outperform these algorithms by sometimes orders of magnitude.

1 Introduction

Dataflow specifications are widespread in areas of digital signal and image processing. In dataflow, a specification consists of a directed graph in which the nodes represent computations and the arcs specify the flow of data. Synchronous dataflow [12] is a restricted form 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., [3, 11, 14]. Typically, code is generated from a given schedule by instantiating inline actor code in the final program. Hence, the size of the required program memory depends on the number of times an actor appears in a schedule, and 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 been published for general SDF graphs [1].

In this paper, we treat the problem of exploring 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 [2].

1.1 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 (and often much greater than) 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 [2], 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 of low polynomial time complexity 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. Experiments show that, although being computationally efficient, these heuristics sometimes produce results that are far from optimal. Even simple test cases may be constructed where the performance (buffer cost) obtained by applying these heuristics differs from the global minimum by more than 2000%, see Example 1.

Example 1. We consider two test graphs and compare different buffer optimization algorithms (see Table 1). The 1st graph with 10 nodes is shown in Fig. 1b). For this simple graph, 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.

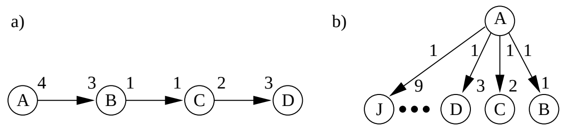

Fig. 1. Simple SDF graphs.

The motivation of the following work was to develop a methodology that is

	Graph 1		Graph 2	
method	best cost (units)	runtime (s)	best cost (units)	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
Exhaust. Search	3003	373	?	?

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

- *Cost-competitive:* the optimization procedure should provide solutions with equal or lower buffering costs as the heuristics APGAN and RPMC in most investigated test cases.
- *Run-time tolerable:* in embedded DSP applications, compilers are allowed to spend more time for optimization of code as in general-purpose compilers, because code-optimality is critical [13].

1.2 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 [2] 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 $\mathcal{O}(N^3)$. The overall picture of the scheduling framework is depicted in Fig. 2.

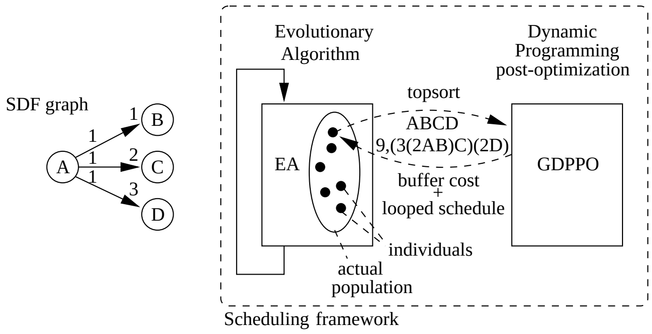

Fig. 2. Overview of the scheduling framework using Evolutionary Algorithms and Dynamic Programming (GDPPO: generalized dynamic programming post optimization for optimally parenthesizing actor orderings [2]) for constructing buffer memory optimal schedules.

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.

2 An Evolutionary Approach for Memory Optimization

2.1 The SDF-scheduling framework

Definition 1 SDF graph. An SDF graph [12] G denotes a 5-tuple $G = (V, A, produced, consumed, delay)$ where
- V is the set of nodes (*actors*) ($V = \{v_1, v_2, \cdots, v_K\}$).
- 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* : $A \to \mathbf{N}$ denotes a function that assigns to each directed arc $\alpha \in A$ the number of produced tokens *produced*(α) per invocation of actor $source(\alpha)$.
– *consumed* : $A \to \mathbf{N}$ denotes a function that assigns to each directed arc $\alpha \in A$ the number of consumed tokens per invocation of actor $sink(\alpha)$.
– *delay* : $A \to \mathbf{N}_0$ 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 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 [12] and captured by a function $q : V \to \mathbf{N}$.

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*.

2.2 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.

The memory requirement is determined by the cost function

$$buffer\_memory(S) = \sum_{\alpha \in A} max\_tokens(\alpha, S), \qquad (1)$$

where $max\_tokens(\alpha, S)$ denotes the maximum number of tokens that accumulate on arc α during the execution of schedule S.[3]

[3] 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 [2].

Example 3. Consider the flat schedule $(\infty(9A)(12B)(12C)(8D))$ for the graph 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 $12 + 12 + 6 = 30$.

2.3 Related Work

The interacion between instruction scheduling and register allocation in procedural language compilers has been studied extensively [9], and optimal management of this interaction has been shown to be intractable [8]. More recently, the issue of optimal storage allocation has been examined in the context of high-level synthesis for iterative DSP programs [5], and code generation for embedded processors that have highly irregular instruction formats and register sets [13, 10]. These efforts do not address the challenges of keeping code size costs manageable in general SDF graphs, in which actor production and consumption parameters may be arbitrary. Fabri [6] 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.

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 [2], 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.

3 Parameterization of the Evolutionary Algorithm

The initial population of individuals, the *phenotype* of which represents a topological sort, is randomly generated. Then, the population iteratively undergoes fitness evaluation (Eq. 1), *selection*, *recombination*, and *mutation*.

3.1 Coding and Repair Mechanism

The optimization problem suggests to use an order-based representation. Each individual encodes a permutation over the set of nodes. As only topological sorts represent legal schedules, a simple repair mechanism transforms a permutation into a topological sort as follows: Iteratively, 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.

Example 5. Recall the SDF graph depicted in Figure 1b), and suppose, the repair algorithm is working on the permutation BCDEFAGHIJ. Since the node A has no ingoing edges but is predecessor of all other nodes, it has to be placed first in any topological sort. The order of the remaining nodes is unchanged. Therefore, the resulting topological sort after the repair procedure is ABCDEFGHIJ.

3.2 Genetic Operators

The selection scheme chosen is *tournament selection*. 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 uniform order-based crossover [4][7], which preserves the permutation property. Mutation is done by permuting the elements between two selected positions, whereas both the positions and the subpermutation are chosen by random (*scramble sublist mutation* [4]).

3.3 Crossover Probability and Mutation Probability

We tested several different combinations of crossover probability p_c and mutation probability p_m on a few random graphs containing 50 nodes.[4]

Based on experimental results, 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. Exemplary, the results for a particular graph are visualized by the 3D plot in Figure 3; 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 4 this is due to premature convergence. The curve

[4] 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.

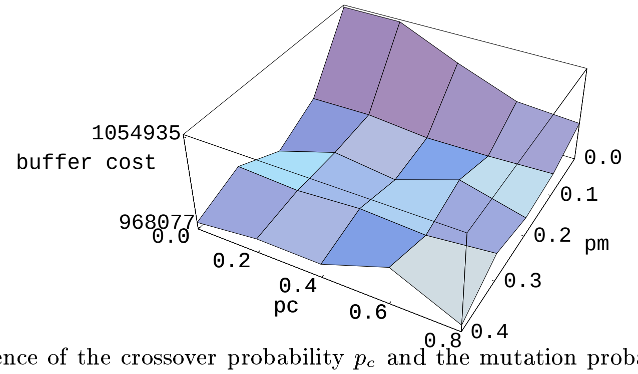

Fig. 3. Influence of the crossover probability p_c and the mutation probability p_m on the average fitness for a particular test graph (3000 fitness evaluations).

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. As a consequence, the Monte Carlo optimization method, that simply generates random points in the search space and memorizes the best solution, might be a better approach to this problem. We investigate this issue in the next section.

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 4 bears out this observation. Beyond it, for this particular test graph a mutation probability of $p_m = 0.2$ and a crossover probability of $p_c = 0$ leads to best performance. This might be interpreted as hint that *Hill Climbing* is also suitable in this domain. The Hill Climbing approach generates new points in the search space by applying a neighborhood function to the best point found so far. Therefore, we also compared the Evolutionary Algorithm to Hill Climbing.

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.

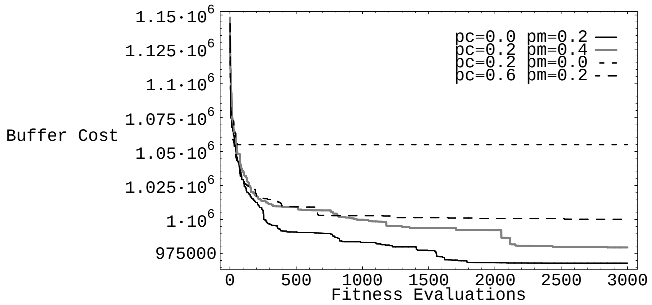

Fig. 4. Performance of the Evolutionary Algorithm according to four different p_c-p_m-combinations; each graph represents the average of ten runs.

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.[6]

4 Experiments

To evaluate the performance of the Evolutionary Algorithm 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, Monte Carlo (MC), and Hill Climbing (HC). We also tried a slightly modified version of the Evolutionary Algorithm 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 [15], whereas the other examples are the same as considered in [2]. 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[5], the search spaces of these problems seem to be rather complex.

In all of the practical benchmark examples that make up Table 2 the results achieved by the Evolutionary Algorithm equal or surpass the ones generated by RPMC. Compared to APGAN on these practical examples, the Evolutionary Algorithm 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 Evolutionary Algorithm. The Monte Carlo simulation, however, performs slightly worse than the other probabilistic approaches.

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

[6] 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.

<	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 Evolutionary Algorithm (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 Evolutionary Algorithm 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.[7] Interestingly, for these graphs APGAN only in 15% of all cases is better than Monte Carlo and only on in two cases better than the Evolutionary Algorithm. On the other hand, it is outperformed by the Evolutionary Algorithm 99% of the time.[8] 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 [2], its better performance (65.5%) is not surprising when directly compared to APGAN. Although, it is beaten by the Evolutionary Algorithm as well as Hill Climbing in 95.5% and 96.5% of the time, respectively.

The obtained results are very promising, but have to be considered in association with their quality, i.e., the magnitude of the buffer costs achieved. In [16], this issue is investigated in detail. In average the buffer costs achieved by the Evolutionary Algorithm are half the costs computed by APGAN and only a fraction of 63% of the 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 Evolutionary Algorithm 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 Evolutionary Algorithm deviate from the costs produced by Hill Climbing by a factor of 0.19% in average).

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

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

5 Conclusions

In summary, it may be said that the Evolutionary Algorithm is superior to both APGAN and RMPC on random graphs. However, both might also be randomized, and thus provide other candidates for comparison, a topic of future research. A comparison with simulated annealing might also be interesting. However, this general optimization method offers many implementation trade-offs such that a qualitative comparison is not possible except under many restricted assumptions.

References

1. S. Bhattacharyya. Compiling data flow programs for digital signal processing. Technical Report UCB/ERL M94/52, Electronics Research Laboratory, UC Berkeley, July 1994.
2. S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell, MA, 1996.
3. 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.
4. Lawrence Davis. *Handbook of Genetic Algorithms*, chapter 6, pages 72–90. Van Nostrand Reinhold, New York, 1991.
5. 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.
6. J. Fabri. *Automatic Storage Optimization*. UMI Research Press, 1982.
7. 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.
8. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, 1979.
9. W.-C. Hsu. Register allocation and code scheduling for load/store architectures. Technical report, Department of Computer Science, University of Wisconsin at Madison, 1987.
10. 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.
11. 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.
12. E.A. Lee and D.G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.
13. P. Marwedel and G. Goossens (eds.). *Code generation for embedded processors*. Kluwer Academic Publishers, Norwell, MA, 1995.
14. 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.
15. 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.
16. J. Teich, E. Zitzler, and S. S. Bhattacharyya. Optimized software synthesis for digital signal processing algorithms - an evolutionary approach. Technical Report 32, TIK, ETH Zurich, Gloriastr. 35, CH-8092 Zurich, January 1998.