Published in: IFIP TC 10 WG 10.5 Workshop on Logic and Architecture Synthesis, pp 220-230, December 18-19, 1995, Grenoble.

A High-Level Synthesis Tool for the Assignment of Storage Values to Sequential Read-Write Memories

Sabih H. Gerez* and Erwin G. Woutersen
Department of Electrical Engineering
University of Twente
email: s.h.gerez@el.utwente.nl

Abstract

Sequential read-write memories (SRWMs) could be used as an alternative to register files in data-path synthesis. They can be considered RAMs without address decoder. A shift register is used instead to sequentially point at memory locations for read or write. Although SRWMs behave similarly to the memory structures recently proposed by Aloqeely and Chen, they are more interesting because of their lower power consumption. Algorithms are presented to check whether a set of storage value fit in a single SRWM (exactly by means of branch-and-bound) and to automatically map storage values in as few SRWMs as possible (by means of heuristics). As opposed to Aloqeely and Chen, also good benchmark results have been obtained for applications with a low degree of "regularity".

1 Introduction

High-level synthesis, the automatic mapping of an algorithmic description of some computation to a description at the register-transfer level, is normally tackled by dividing the problem into a number of subproblems and solving them sequentially (see e.g. [4, 6]). These subproblems include *scheduling* and *assignment*. Scheduling maps an operation in the algorithmic description to a time instant and assignment deals with the mapping to computational units, storage units and interconnection. Most of the time the scheduling problem is solved prior to the assignment problem (although successful approaches based on the reverse order have been reported, see e.g. [20]).

This paper addresses the memory assignment problem for the case that a schedule has been fixed. In general, it will be necessary to store intermediate results in some way in order to

*The work reported in this document was initiated when the author was an *academic guest* at the Institute of Microtechnology, Neuchâtel, Switzerland in the months June and July of 1994. The author is grateful to the Ecole Polytechnique Fédérale de Lausanne, Switzerland for providing the financial support for the visit.

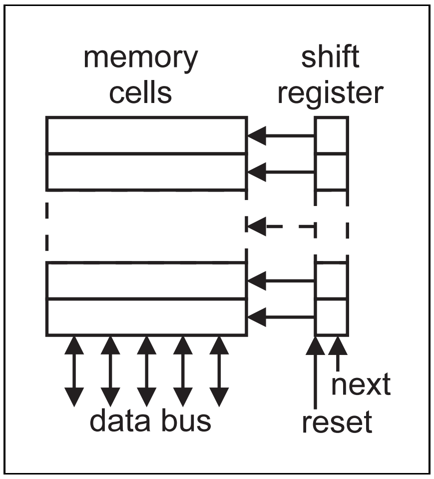

Figure 1: *A sequential memory.*

satisfy the result of scheduling. These intermediate results are called *storage values* [21] and it is part of the assignment task to map these values on memory elements available within an architectural model.

Normally *registers* and *register files* are used for the purpose of storage in a data path (see e.g. [8]). Registers have a single memory location while register files have multiple locations identified by an address (similar to a RAM). Recently, *queues*, *stacks*, and *bidirectional queues* have been proposed as alternative memory elements by Aloqeely and Chen [1, 2, 3]. They consist of structures of shift register chains in which the writing and reading of data occurs in first-in-first-out order (queue: write in first register and read from last register in chain), last-in-first-out order (stack: write to and read from first register only) or a combination of both (bidirectional queue: write to or read from either the first or the last register in the chain).

This paper considers yet another type of memory element: the *sequential read-write memory* (SRWM). It can be considered as "RAM without address decoding". Instead of an address decoder there is a shift register. Only one of the bits in this register is high and it is used as a pointer to a memory location to which data can be written or from which data can be read[1]. A control signal (`next`) can make this bit point to the next higher memory location in the next control step and another control signal (`reset`) can reset the shift register such that the high bit points at the lowest memory location. This has been visualized in Figure 1. These memories have been designed and used by Heubi et al. [11, 12]. Obviously, the read-only version of such a memory is well suited for the storage of multiplication coefficients in the realization of DSP algorithms characterized by the (almost) infinite repetition of the same algorithm on a stream of data (in subsequent iterations the same coefficients need to be fed into a multiplier in the same order). The read-only version can also be adapted for the realization of controllers [13].

The use of a sequential memory for the storage of intermediate values produced and consumed in the data path is far less obvious. Normally *random access memories* are used for this purpose,

[1]The actual (cheaper) hardware used for addressing is slightly different [10, 22].

which allows the handling of read and write requests for any memory location. Sequential memories are far more constrained in this respect. The reason that they might still be considered to be used as a *read-write memory*, is that they are simpler: as there is no address decoding, only few control signals are necessary (instead of an address) which also leads to a simplification of the controller. SRWMs have a similar behavior as the structures proposed by Aloqeely and Chen, but are more interesting as they are generally smaller and use less power (the data is, for example, not moved around in every control step). A study comparing SRWM and shift-register structures for area and power consumption has been reported by Heubi [10, 11]; it turns out that SRWMs are better in both aspects once that the memories are sufficiently large (in word length and number of memory locations).

The memory elements discussed in this paper should be seen as an alternative to memory elements that are traditionally used. We do not claim that they are preferable in all circumstances. The tool described here should therefore be seen as part of a toolbox that also contains synthesis tools for other memory types.

Storage values that do not have read or write conflicts can be stored in a single RAM without bothering about the address at which they will reside: all addresses are equally accessible. This is not the case for an SRWM: the read and write times of storage values restrict the address to which a specific value can be mapped. The first problem discussed in this paper deals with the problem of determining whether a specific set of storage values fits in a single SRWM and providing the address for each storage value when there is a fit. The second problem is to partition an arbitrary set of storage values in a minimal number of subsets such that each subset can be mapped on a single SRWM. A secondary goal is to minimize the total of number memory locations used. The first problem is exactly solved in this paper by a branch-and-bound algorithm. This algorithm is later used as a subroutine of a heuristic that solves the second problem.

The structure of this paper is as follows. First two combinatorial optimization problems related to the use of SRWMs in high-level synthesis are formulated. Then algorithms are proposed to solve these problems. The final part of the paper consists of the presentation of experimental results obtained by these algorithms and some conclusions.

2 Formulation of the Problems

The application domain addressed in this paper is digital signal processing, characterized by the (almost) infinite iteration of some algorithm. Such an algorithm is characterized by a parameter T_0 called the *iteration period* [9] or *data initiation interval* [15]. All computations are repeated every T_0 control steps, where a control step corresponds to a single period of a system clock. This does not mean that all operations belonging to a single execution of an algorithm need to be performed within T_0 control steps. Assuming an *overlapped scheduling* [19] or *loop folding* [15] model, the *schedule span* can be larger than T_0.

One iteration period covers the control steps $T = \{0, 1, \ldots, T_0 - 1\}$. The result of a scheduling algorithm, containing the mapping in time of a single iteration, will, however, have references

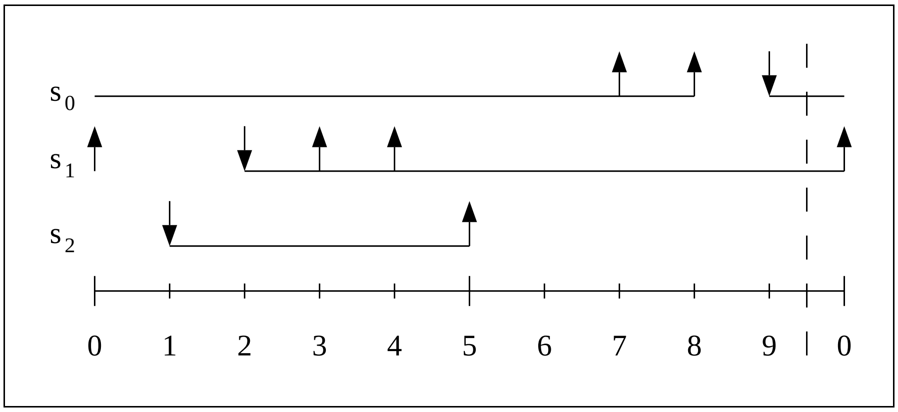

Figure 2: *A set of storage values.*

to control steps in Z (the set of integers). The storage values will be part of the set $S = \{s_0, s_1, s_2, \ldots\}$. For each value $s \in S$, $\omega(s)$ is the write time, the time when the value should be stored in memory: $\omega : S \rightarrow Z$. A value can be read more than once after having been written. The time instances at which this happens are indicated by the function $\rho : S \rightarrow 2^Z$, where 2^Z is the *power set* of Z (the set of all its subsets).

The latest read time for a storage value is given by the function $\epsilon : S \rightarrow Z$: $\epsilon(s) = \max_{t \in \rho(s)}$. The *lifetime interval* [21] of a value s is then defined by $[\omega(s), \epsilon(s)]$. The *lifetime* $\lambda(s)$ of a storage value s is defined as: $\epsilon(s) - \omega(s)$.

Even when the values of $\omega(s)$ and the members of $\rho(s)$ may not always be in the set T, it is useful for the rest of the analysis to concentrate on time values in T. Therefore, the auxiliary functions $\omega_m : S \rightarrow T$ and $\rho_m : S \rightarrow 2^T$ are introduced: $\omega_m(s) = \omega(s) \bmod T_0$ and $\rho_m(s) = \{t \bmod T_0 | t \in \rho(s)\}$. In a similar way, $\epsilon_m : S \rightarrow T$ and $\epsilon_m(s) = \epsilon(s) \bmod T_0$. Figure 2 shows an example of a set of storage values after taking all time values modulo T_0 ($T_0 = 10$). An arrow pointing downward indicates the write time of a storage value and an arrow pointing upward a read time.

For the time being, it is assumed that all storage values $s \in S$, have a lifetime of at most T_0. In case of storage values with an actual lifetime longer than T_0, they are supposed to have been split into values with a lifetime of at most T_0 in a preprocessing step.

The following combinatorial optimization problems with respect to synthesis with sequential read-write memories are considered here:

1. REALIZABILITY. Given a set S of storage values and a set T of control steps, find out whether all values can be stored in a single SRWM. If so, indicate the address to which each storage value should be assigned.

2. MINIMAL GROUPING. Given sets S and T, find a partition of S in disjoint subsets S_1, S_2, …, S_k, such that each subset can be realized on a separate SRWM and that k is minimal.

Algorithms for both problems have been developed. They are presented in the next two sections.

3 An Exact Solution for the Realizability Problem

Obviously, for any optimization problem finding an exact solution is often more desirable than having an approximation. Because the realizability problem is a central problem in SRWM synthesis, we have chosen to solve it exactly. As the problem is NP-complete [22], it is solved by a branch-and-bound algorithm as explained below.

An important observation in searching for the solution of the realizability problem is that the decision to assign a storage value to a specific address location can have consequences for the assignments possible for other storage values. Suppose that $A = \{0, 1, \ldots\}$ is the set of possible location addresses (note that the size of this set is bounded by $|S|$). Consider the function $\gamma : S \times A \times S \rightarrow 2^A$. $\gamma(s_i, k, s_j)$ gives the set of possible addresses for s_j, once that a (tentative) decision has been made to assign s_i to address k. Only those storage values s_j are considered that have a read or write action immediately before or after a read or write action of s_i, where "immediate" means that no third storage values has a read or write action in the mean time. In such a case s_i and s_j will be called *neighbor* storage values. $\gamma(s_i, k, s_j) = \emptyset$ when s_i and s_j are not neighbors. γ is computed by looking at the number of control steps between actions of neighbors. In this available time, the address pointer may or may not be reset to zero (with the `reset` signal) and it can be advanced at most as many times as there are control steps (with the `next` signal). If s_i and s_j have nonoverlapping lifetimes, $k \in \gamma(s_i, k, s_j)$ (they can be stored at the same location).

In the example of Figure 2, $\gamma(s_2, 0, s_1) = \{1\}$, because s_2 is written at time 1 and s_1 at time 2, a decision to fix s_2 at address 0 forces s_1 at address 1 (there is only time for one "next" signal); the requirement arising from the read action of s_1 at time 0 (4) followed by the write (read) action on s_2 at time 1 (5), is less strict and would allow both addresses 1 and 2 for s_1 (as a reset signal sets the address pointer to 0 in one control step). Reasoning in the same way $\gamma(s_2, 0, s_0) = \{1, 2\}$, because nothing happens at time 6, giving the opportunity for two "next" signals between control steps 5 and 7.

It is not difficult to see that γ is well-suited for a backtracking search procedure based on branch-and-bound [14]. For each storage value for which more than one address assignment is possible, a tentative choice for one of the addresses is made. The other choices will be possibly considered after backtracking. The possible address locations for the other storage values are restricted by the application of γ before continuing the recursive search process. A solution has been found when all storage values have a single possible address assignment that obeys γ. If a situation occurs where some storage value cannot be assigned to any location, no solution can be found by further searching and backtracking can start directly. The same is true when it can be estimated from some situation that the number of locations needed will be at least as many as in the best solution found that far.

Specific combinations of read-write patterns can directly lead to the conclusion that no solution

```
maxmem := 0;
for all s ∈ S "in random order"
do
   flag := true;
   for i := 1 to maxmem do
     if realizable({s} ∪ R_i) and flag
     then R_i := {s} ∪ R_i;
           flag := false;
     fi
   if flag
   then maxmem := maxmem + 1;
         R_maxmem := {s}
   fi
od;
```

Figure 3: *A simple algorithm for initial assignment in minimal grouping.*

exists or can put initial constraints on address locations before starting branch-and-bound search. These patterns are mentioned in [7] and are further generalized in [22]. They are part of the implementation of which experimental results are mentioned later on.

The reader should check that the problem of Figure 2 is realizable with the unique solution that maps s_0 to address 2, s_1 to address 0 and s_2 to address 1.

4 A Heuristic Approach for the Minimal Grouping Problem

The fact that the investigation whether a set of storage values is realizable on a single SRWM is already quite complex, makes that an exact solution of the minimal grouping problem is not feasible. Obviously, any heuristic that attempts to solve the minimal grouping problem, will need to use an algorithm for the realizability (at least to check whether a given grouping is correct). As our realizability algorithm has an exponential worst-case time complexity, we had to find an effective heuristic that required a "limited" use of the realizability check (search methods that visit many potential solutions, like simulated annealing, could not be considered).

The algorithm that we propose for the minimal grouping problem consists of two stages both of which use the routine for realizability test. The first is an initial assignment phase in which a solution with some number of memories is constructed such that the assignments to all memories is realizable. The pseudocode for a simple algorithm solving this task is given in Figure 3. After a random ordering of the storage values, it takes each storage value and tests whether the value can be mapped on one of the memories already allocated. If not, a new memory is allocated to hold this storage value. The set of storage values assigned to a memory

```
repeat
  P := R_1;
  for i := 1 to maxmem – 1
  do R_i := R_{i+1}
  od;
  maxmem := maxmem – 1;
  for all s ∈ P
  do
     flag := true;
     for i := 1 to maxmem do
        if realizable({s} ∪ R_i) and flag
        then R_i := {s} ∪ R_i;
               flag := false;
        fi
     if flag
     then maxmem := maxmem + 1;
            R_maxmem := {s}
     fi
  od
until "loop condition detected";
```

Figure 4: *The iterative improvement algorithm for minimal grouping.*

with index i is denoted by R_i.

The second stage is an iterative improvement stage. Its pseudocode description is given in Figure 4. The main idea is to take the set P of all storage values in the first memory and try to assign them to any of the remaining memories. If this does not succeed for some storage value, this value ends up in the newly created "overflow" memory. As the set of storage values that is being redistributed is known to be realizable, the worst that can happen is that all its elements end up in the overflow memory. So, the number of memories never increases, but may decrease.

The process just describes is repeated while being careful not to enter an infinite loop. If the set P at a certain iteration equals any of the sets P encountered in an earlier iteration, the iterative improvement process is terminated. During this search process the solution with the minimal number of total memory locations (for the optimal number of memories) is recorded as the best solution.

Finally, there is a postprocessing step that attempts to decrease the total number of memory locations of the best solution. It exhaustively searches for the possibility to redistribute storage values mapped on the same location of some memory without increasing the number of memory locations of the other memories. Each time this succeeds, the gain is one memory location.

Storage values whose lifetime $\lambda(s)$ are bigger than T_0 are split following the following heuristic: create as many storage values with a lifetime of $T_0 - 1$ until the lifetime of the remaining value is at most T_0. Note that $T_0 - 1$ is used instead of T_0 in order to avoid that the newly created storage values have read or write conflicts with each other.

5 Experimental Results

The algorithms mentioned above have been implemented in CMU Common Lisp on an HP 735 machine. The implementation can handle a hardware model in which a clock cycle has separate read and write phases (allowing the reuse in the same clock cycle of a memory location after a last read) and a model in which no distinct phases exist. The results reported here are based on the first model. It turns out that good results are also obtained for benchmark examples that would have been qualified as "irregular" by Aloqeely and Chen [1, 2, 3] (their algorithms are based on "clustering" groups of storage values with equal lifetimes and similar methods).

A problem with benchmarks for memory synthesis is that the problem is often solved as part of a complete high-level synthesis problem. Storage value lifetimes are taken from the specific scheduler used and not seen as a separate benchmark. We have applied the minimal grouping algorithm to the set of storage values generated by an in-house scheduling and assignment tool [17] for the well-known "fifth-order wave-digital filter" benchmark [5], a benchmark not considered by Aloqeely and Chen due to its "irregularities". The specific instance of the problem had 20 storage values for which it can be shown that an implementation with common RAMs would require at least 4 memories. 100 different runs of the program (to be less dependent on a specific random choice in the initial assignment phase) all gave a solution with 4 SRWMs and a total number of address locations varying between 8 and 11. The average CPU runtime was 9.9 seconds.

A family of benchmark examples that do not directly originate from a scheduler have been proposed by Parhi [18]. They consist of the repetitive transpositions of arbitrary size matrices. Consider e.g. the following 3×4 matrix:

$$\begin{bmatrix} s_0 & s_1 & s_2 & s_3 \\ s_4 & s_5 & s_6 & s_7 \\ s_8 & s_9 & s_{10} & s_{11} \end{bmatrix}$$

of which the elements arrive in a row-by-row order (one element per control step): s_0 followed by s_1, s_2, s_3, s_4, etc. Transposition requires that the storage values are output in a column-by-column order: s_0, followed by s_4, s_8, s_1, s_5, etc. Once that all elements of one matrix have been processed, a new matrix to be transposed arrives. Obviously, $T_0 = nm$ for an $n \times m$ matrix. Parhi shows that for these problems $(n-1)(m-1)$ is a lower bound for the number of memory locations (and that this lower bound can be met using shift register structures interconnected in a specific way).

These benchmarks have the property that they contain storage values with lifetimes longer than T_0. They have been split in the way mentioned in Section 4. An implementation with RAMs would also require splitting, unless special addressing hardware is used [16]. Table 1 shows

size	# memories			# locations			CPU time
	min	max	lb	min	max	lb	[sec]
2×8	3	3	2	10	12	7	8.15
2×10	3	3	3	12	14	9	15.9
3×3	2	2	2	6	6	4	0.46
3×4	3	3	2	8	9	6	3.66
3×5	3	4	3	11	13	8	7.19
3×6	3	4	3	13	14	10	16.9
4×4	3	3	3	10	12	9	14.2
4×5	3	4	3	15	17	12	30.4
5×5	3	4	3	17	19	16	159
6×6	4	4	3	26	27	25	382

Table 1: *Experimental results for matrix transposition benchmarks.*

some results obtained for different matrix dimensions. For each problem several runs have been performed to be less dependent on the influence of randomization (ranging from 100 runs for the smaller problems down to 5 for the largest one). The given CPU time is the average per run. Roughly said (there is quite some variation for different benchmarks), the time spent on initial assignment is 50 %, the time spent on iterative improvement is 40 %, while about 10 % of the time is spent on postprocessing. The minimal and maximal number of memories found is listed in the table. The column "lb" for memories gives a "hand computed" lower bound for the case that RAMs instead of SRWMs would be used for the given splitting. Also the minimal and maximal total number of location found for each case is listed. The "lb" column here refers to the lower bound in the hardware model of Parhi (see above); it is not likely that the lower bound can be realized in a hardware model of RAMs or SRWMs.

6 Conclusions and Further Research

This paper has shown that SRWMs deserve to be considered as an alternative for the storage of values in data-path synthesis. The results found for the matrix transposition benchmarks indicate that their use does not lead to serious hardware inefficiencies in spite of the limited addressing capabilities of SRWMs. A tool as the one presented here could therefore be incorporated in a toolbox for high-level synthesis.

Two algorithms were presented to automatically map storage values on SRWMs. The first one solves the realizability problem exactly using an algorithm that has an exponential worst-case time complexity. The second one uses the first as a subroutine to solve the minimal-grouping problem in a heuristic way. It, therefore, also has an exponential worst-case time complexity. Short-term future research will concentrate on speeding up these algorithms, possibly by relaxing the requirement of solving the realizability problem exactly.

Issues for longer-term research are investigating a more clever way of splitting storage values that have a lifetime longer than T_0, considering variations on the SRWM model presented here and paying more attention to theoretical matters as the complexity class of the problem.

The ultimate issue is to integrate an SRWM-synthesis tool within a larger framework of tools for data-path synthesis, especially a synthesis system for low-power design.

References

[1] M. Aloqeely. *Sequencers: A New Alternative for Data Path Synthesis*. PhD thesis, Department of Electrical and Computer Engineering, Syracuse University, June 1995.

[2] M. Aloqeely and C.Y.R. Chen. A new technique for exploiting regularity in data path synthesis. In *European Design Automation Conference, EURO-DAC*, pages 394–399, 1994.

[3] M. Aloqeely and C.Y.R. Chen. Sequencer-based data path synthesis of regular iterative algorithms. In *31st ACM/IEEE Design Automation Conference*, pages 155–160, 1994.

[4] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York, 1994.

[5] P. Dewilde, E. Deprettere, and R. Nouta. Parallel and pipelined VLSI implementations of signal processing algorithms. In S.Y. Kung, H.J. Whitehouse, and T. Kailath, editors, *VLSI and Modern Signal Processing*, pages 257–276. Prentice Hall, Englewood Cliffs, New Jersey, 1985.

[6] D.D. Gajski, N.D. Dutt, A.C.H. Wu, and S.Y.L. Lin. *High-Level Synthesis, Introduction to Chip and System Design*. Kluwer Academic Publishers, Boston, 1992.

[7] S.H. Gerez. Combinatorial optimization problems related to the use of sequential read-write memories in high-level synthesis. Technical Report EL-BSC-079N94, University of Twente, Department of Electrical Engineering, December 1994.

[8] B.S. Haroun and M.I. Elmasry. Architechtural synthesis for DSP silicon compilers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(4):431–447, April 1989.

[9] S.M. Heemstra de Groot, S.H. Gerez, and O.E. Herrmann. Range-chart-guided iterative data-flow-graph scheduling. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 39:351–364, May 1992.

[10] A. Heubi. Architecture faible consomation pour le traitement numérique du signal. Technical Report 330 PE 10/92, Institute of Microtechnology, University of Neuchâtel, October 1992.

[11] A. Heubi, M. Ansorge, and F. Pellandini. Architecture VLSI faible consommation pour le traitement numérique du signal. In *Proceedings GRETSI '93*, pages 1083–1086, Juan-les-Pins, France, September 1993.

[12] A. Heubi, S. Grassi, M. Ansorge, and F. Pellandini. A low power VLSI architecture with an application to adaptive algorithms for digital hearing aids. In M.J.J. Holt, C.F.N. Cowan, P.M. Grant, and W.A. Sandham, editors, *Signal Processing VII: Theories and Applications (Proceedings of the EUSIPCO-94 Seventh European Signal Processing Conference)*, pages 1875–1878, 1994.

[13] H.A. Hilderink and J.A.G. Jess. Rom-based multi thread controller. In *IFIP Workshop on Logic and Architecture Synthesis*, pages 231–241, Grenoble, December 1993.

[14] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, Rockville, Maryland, 1978.

[15] T.F. Lee, A.C.H. Wu, Y.L. Lin, and D.D. Gajski. A transformation-based method for loop folding. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):439–450, April 1994.

[16] J.L. van Meerbergen, P.E.R. Lippens, W.F.J. Verhaegh, and A. van der Werf. Relative location assignment for repetitive schedules. In *European Conference on Design Automation with the European Event on ASIC Design, EDAC/EUROASIC*, pages 403–407, 1993.

[17] A. Oláh, S.H. Gerez, and S.M. Heemstra de Groot. Scheduling and allocation for the high-level synthesis of DSP algorithms by exploitation of data transfer mobility. In *International Conference on Computer Systems and Software Engineering, CompEuro 92*, pages 145–150, May 1992.

[18] K.K. Parhi. Systematic synthesis of DSP data format converters using life-time analysis and forward-backward register allocation. *IEEE Transactions on Circuit and Systems - II: Analog and Digital Signal Processing*, 39(7):423–440, July 1992.

[19] K.K. Parhi and D.G. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. *IEEE Transactions on Computers*, 40(2):178–195, February 1991.

[20] M. Potkonjak and J.M. Rabaey. Scheduling algorithms for hierarchical data control flow graphs. *International Journal of Circuit Theory and Applications*, 20:217–233, 1992.

[21] L. Stok and J.A.G. Jess. Foreground memory management in data path synthesis. *International Journal of Circuit Theory and Applications*, 20:235–255, 1992.

[22] E.G. Woutersen. The application of sequential read-write memories in high-level synthesis. Master's thesis, University of Twente, Department of Electrical Engineering, Laboratory of Network Theory, December 1995. In preparation.