3D Exploration of Software Schedules for DSP Algorithms

J. Teich
Computer Engineering
University of Paderborn
Germany

E. Zitzler
Institute TIK
ETH Zürich
Switzerland

S. S. Bhattacharyya
University of Maryland
College Park
U.S.A.

Abstract

This paper addresses the problem of exploring trade-offs between *program memory*, *data memory* and *execution time* requirements (3D) for DSP algorithms specified by data flow graphs. Such an exploration is of utmost importance for being able to analyze the feasibility and range of possible software solutions as part of a *hardware/software codesign methodology* where the target processor and the code generation style may lead to complete different solutions of the same specification. For solving this multi-objective optimization problem, an *Evolutionary Algorithm* approach is applied. In particular, a new *Pareto-optimization* algorithm is introduced. For different well-known target DSP processors, the Pareto-fronts are analyzed and compared.

1 Introduction

Here, we study the effects between instantiating code from data flow graph specifications by inlining or subroutine calls as well as the effect of loop nesting and context switching on a target processor (DSP) that is used as a component in a memory and cost-critical hw/sw-design, e.g., a single-chip solution. We present the first systematic optimization framework for exploring implementation trade-offs in the 3-dimensional run-time/program memory/data memory space of processor schedules.

The methodology begins with a given *synchronous dataflow graph* [5] as used in many rapid prototyping environments as input for code generators for programmable digital signal processors (PDSPs) [3, 4, 6].

Example 1 *A practical example is a sample-rate conversion system. In Fig. 1, a digital audio tape (DAT), operating at a sample rate of 48 kHz is interfaced to a compact disk (CD) player operating at a sampling rate of 44.1 kHz, e.g., for recording purposes, see [9] for details on multistage sample rate conversion.*

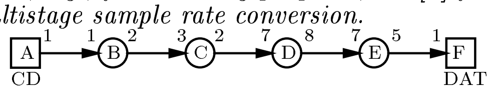

Figure 1: CDtoDAT conversion benchmark

As reported by DSP analyists (e.g., the DSPStone bencharking group [11]), today's DSP compilers still produce several 100%s of overhead with respect to assembly code written and optimized by hand. A commonly used approach in SDF-based design environments that avoids the limitations of current compiler technology is to store optimized assembly code for each actor in a target-specific library and to generate code from a given schedule by instantiating actor code in the final program. By doing this, the influence of the compiler technology may be taken out as one unknown factor of efficiency.

Prior work on code size minimization of SDF schedules has focused on an inline code generation model [1]. The total memory requirement may then be approximated by a linear combination of the (weighted) number of actor appearances in a schedule. Evidently, so called *single appearance schedules* (SASs), where each actor appears only once in a schedule, are program memory optimal under this model. However, they may not be data memory minimal, and in general, it may be desirable to trade-off some of the run-time efficiency of code inlining with further reduction in code size by using subroutine calls, especially with system-on-a-chip implementations. For example, if only a very small amount of program memory is available for a signal processing subsystem, but the data memory and speed constraints are not tight, then a compact looped schedule organization with heavy use of subroutines would be desirable. Similarly, if the data memory and execution time are severe "bottlenecks", but program space is abundant, then a minimal buffer schedule organization (which typically precludes the use of extensive looping [2] with inline code) would be preferable. Here, we seek to exploit the dimensions of program memory, data memory, and execution time requirements simultaneously.

2 SDF Scheduling Framework

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

- *V is the set of nodes (*actors*) ($V = \{v_1, v_2, \cdots, v_{|V|}\}$).*
- *A is the set of directed arcs. With $source(\alpha)$ ($sink(\alpha)$), we denote the source (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 $delay(\alpha)$ that reside on α.*

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*.

Example 2 *For the CDtoDAT graph in Figure 1, the minimal number of actor firings is obtained as* $q(A) = q(B) = 147$, $q(C) = 98$, $q(D) = 28, q(E) = 32, q(F) = 160$. *The schedule* $(\infty(7(7(3AB)(2C))(4D))(32E(5F)))$ *represents a valid schedule.*

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*, or simply SAS in the following, if it contains only one appearance of each actor.

Example 3 *The schedule* $(\infty(147A)(147B)(98C)(28D)(32E)(160F))$ *is a valid SAS for the graph shown in Fig. 1.*

2.1 Code generation model

For each actor in a valid schedule S, we insert a code block that is obtained from a library of predefined actors or a simple subroutine call of the corresponding subroutine, and the resulting sequence of code blocks (and subroutine calls) is encapsulated within an infinite loop to generate a software implementation. Each schedule loop thereby is translated into a loop in the target code.

Example 4 *For the simple SDF graph in Fig. 2a), a buffer model for realizing the data buffer on the arc* α *as well as a pseudo assembly code notation (similar to the Motorola DSP56k assembly language) for the complete code for the schedule* $S = (\infty(3A)(2B))$ *is shown in Fig. 2b), c) respectively. There is a location loc that is the address of the first memory cell that implements the buffer and one read (*$rp(\alpha)$*) and write pointer (*$wp(\alpha)$*) to store the actual read (write) location. The notation*

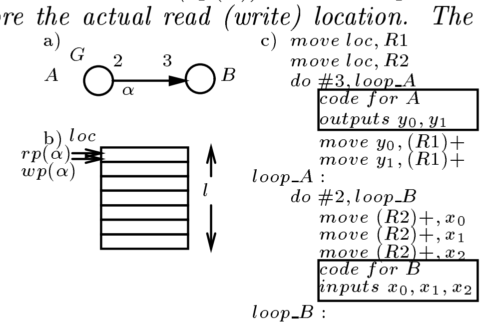

Figure 2: SDF graph a), memory model for arc buffer b), and Motorola DSP56k-like assembly code realizing the schedule $S = (\infty(3A)(2B))$.

do #N LABEL denotes a statement that specifies N successive executions of the block of code between the do-statement and the instruction at location LABEL. First, the read pointer $rp(\alpha)$ *to the buffer is loaded into register* $R1$ *and the write pointer* $wp(\alpha)$ *is loaded into* $R2$. *During the execution of the code, the new pointer locations are obtained without overhead using autoincrement modulo addressing* $((R1)+, (R2)+)$. *For the above schedule, the contents of the registers (or pointers) is shown in Fig. 3.*

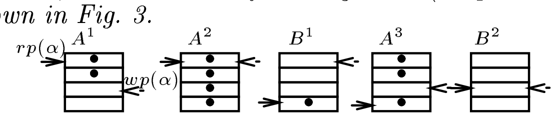

Figure 3: Memory accesses for the schedule $S = (\infty A(2AB))$.

3 Optimization Metrics

3.1 Program memory overhead $P(S)$

Assume that each actor N_i in the library has a program memory requirement of $w(N_i) \in \mathbf{N}$ memory words. Let $flag(N_i) \in \{0, 1\}$ denote the fact whether in a schedule, a subroutine call is instantiated for all actor invocations of the schedule ($flag(N_i) = 0$) or whether the actor code is inlined into the final program text for each occurence of N_i in the code ($flag(N_i) = 1$). Hence, given a schedule S, the program memory overhead $P(S)$ will be accounted for by the following equation:[1]

$$\begin{aligned} P(S) &= \sum_{i=1}^{|V|} (app(N_i, S) \cdot w(N_i) \cdot flag(N_i)) \\ &+ (w(N_i) + app(N_i, S) * P_S) \cdot (1 - flag(N_i)) \\ &+ P_L(S) \qquad (1) \end{aligned}$$

In case one subroutine is instantiated ($flag(N_i) = 0$), the second term is non-zero adding the fixed program memory size of the module to the cost and the subroutine call overhead P_S (code for call, context save and restore, and return commands). In the other case, the program memory of this actor is counted as many times as it appears in the schedule S (inlining model). The additive term $P_L(S) \in \mathbf{N}$ denotes the program overhead for looped schedules. It accounts for a) the additional program memory needed for loop initialization, and b) loop counter incrementation, loop exit testing and branching instructions. This overhead is processor-specific, and in our computations proportional to the number of loops in the schedules.

3.2 Buffer memory overhead $D(S)$

We account for overhead due to data buffering for the communication of actors. The simplest model for buffering is to assume that a distinct segment of memory is allocated for each arc of a given graph.[2] The amount of data needed to store the tokens that accumulate on each arc during the evolution of a schedule S is given as:

$$D(S) = \sum_{\alpha \in A} max\_tokens(\alpha, S) \qquad (2)$$

[1] $app(N_i, S)$: number of times, N_i appears in the schedule string S).

[2] In [8], we introduced different models for buffer sharing, and efficient algorithms to compute buffer sharing. Due to space requirements, and for matters of comparing our approach with other techniques, we use the above simple model here.

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

Example 5 *Consider the schedule in Example 3 of the CDtoDAT benchmark. This schedule has a buffer memory requirement of* $1471+1472+982+288+325=1021$. *Similarly, the buffer memory requirement of the looped schedule* $(\infty(7(7(3AB)(2C))(4D))(32E(5F)))$ *is* 264.

3.3 Execution Time Overhead $T(S)$

With execution time, we denote the duration of execution of one iteration of a SDF graph comprising $q(N_i)$ activations of each actor N_i in clock cycles of the target processor.[3]

In this work, we account for the effects of (1) loop overhead, (2) subroutine call overhead, and (3) buffer (data) communication overhead in our characterization of a schedule. Our computation of the execution time overhead of a given schedule S therefore consists of the following additive components:

Subroutine call overhead: For each instance of an actor N_i with $flag(N_i) = 0$, we add a processor specific latency time $L(N_i) \in \mathbf{N}$ to the execution time. This number accounts for the number of cycles needed for storing the necessary amount of context prior to calling the subprogram (e.g., compute and save incremented return address), and to restore the old context prior to returning from the subroutine (sometimes a simple branch).[4]

Communication time overhead: Due to static scheduling, the execution time of an actor may be assumed fixed (no interrupts, no I/O-waiting) necessary), however, the time needed to communicate data (read and write) depends in general a) on the processor capabilities, e.g., some processors are capable of managing pointer operations to *modulo buffers* in parallel with other computations.[5], and b) on the chosen buffer model (e.g., contiguous versus non-contiguous buffer memory allocation). In a first approximation, we define a penalty for the read and write execution cycles that is proportional to the number of data read (written) during the execution of a schedule S. For example, such a penalty may be of the form

$$IO(S) = 2 \sum_{\alpha=(N_i,N_j)\in A} q(N_i) produced(N_i) T_{io} \qquad (3)$$

where T_{io} denotes the number of clock cycles that are needed between reading (writing) 2 sucessive input (output) tokens.

Loop overhead: For looped schedules, there is in general the overhead of initializing and updating a loop counter, and of checking the loop exit condition, and of branching, respectively. The loop overhead for one iteration of a simple schedule loop L (no inner loops contained in L) is assumed a constant $T_L \in \mathbf{N}$ of processor cycles, and its initialization overhead $T_L^{init} \in \mathbf{N}$. Let $x(L) \in \mathbf{N}$ denote the number of loop iterations of loop L, then the loop execution overhead is given by $O(L) = T_L^{init} + x(L) \cdot T_L$. For nested loops, the total overhead of an innermost loop is given as above, whereas for an outer loop L, the total loop overhead is recursively defined as

$$O(L) = T_L^{init} + x(L) \cdot \left(T_L + \sum_{L' \text{ evoked in} L} O(L') \right) \qquad (4)$$

The *total loop overhead* $O(S)$ of a looped schedule S is the sum of the loop overheads of the outermost loops.

Example 6 *Consider the schedule* $(\infty(3(3A)(4B))(4(3C)(2D)))$, *and assume that the overhead for one loop iteration* $T_L = 2$ *cycles in our machine model, the initialization overhead being* $T_L^{init} = 1$. *The outermost loop consists of 2 loops* L_1 *(left) and* L_2 *(right). With* $O(S) = 1 + 1 \cdot (2 + O(L_1) + O(L_2))$ *and* $x(L_1) = 3$, $x(L_2) = 4$, *we obtain the individual loop overheads as* $O(L_1) = 1 + 3 \cdot (2 + O(3A) + O(4B))$ *and* $O(L_2) = 1 + 4 \cdot (2 + O(3C) + O(2D))$. *The innermost loops* $(3A)$, $(4B)$, $(3C)$, $(2D)$ *have the overheads* $1+6, 1+8, 1+6, 1+4$, *respectively. Hence,* $O(L_1) = 1 + 3 \cdot 18$ *and* $O(L_2) = 1 + 4 \cdot 14$, *and* $O(S)$ *becomes* 115 *cycles.*

In total, $T(S)$ of a given schedule S is defined as

$$\begin{aligned} T(S) &= \left(\sum_{i=1}^{|V|} (1 - flag(N_i)) \cdot L(N_i) \cdot q(N_i)\right) \\ &+ \; IO(S) \; + \; O(S) \end{aligned} \qquad (5)$$

Example 7 *Consider again Example 6. Let the individual execution time overheads for subroutine calls be* $L(A) = L(B) = 2$, *and* $L(C) = L(D) = 10$ *cycles. Furthermore, let code for* A *and* C *be generated by inlining* ($flag(A) = flag(C) = 1$) *and by subroutine call for the other actors. Hence,* $T(S) = L(B) \cdot q(B) + L(D) \cdot q(D) + O(S) + IO(S)$ *results in* $T(S) = 2 \cdot 12 + 10 \cdot 8 + 115 + IO(S) = 219 + IO(S)$. *Hence, the execution overhead is* 219 *cycles with respect to the same actor execution sequence but with only inlined actors and no looping at all.*

3.4 Target processor modeling

For the following experiments, we will characterize the influence of a chosen target processor by the following overhead parameters using the above target (overhead) functions:

- P_S: subroutine call overhead (number of cycles) (here: for simplicity assuming independence of actor, and no context to be saved and restored except PC and status registers).

[3] Note that this measure is equivalent to the inverse of the throughput rate in case it is assumed that the outermost loop repeats forever.

[4] Note that the exact overhead may depend also on the register allocation and buffer strategy. Furthermore, we assume that no nesting of subroutine calls is allowed. Also, recursive subroutines are not created and hence disallowed. Under these conditions, the context switching overhead will be approximated by a constant $L(N_i)$ for each module N_i or even to be a processor-specific constant T_S, if no information on the compiler is available. Then, T_S may by chosen as an average estimate or by the worst-case estimate (e.g., all processor registers must be saved and restored upon a subroutine invocation).

[5] Note that this overhead is then highly dependent on the register allocation strategy.

- P_L: the number of program words for a complete loop instruction including initialization overhead.
- T_S: the number of cycles required to execute a subroutine call and a return instruction and to store and recover context information.
- T_L, T_L^{init}: loop overhead, loop initialization overhead, respectively in clock cycles.

Three real DSPs have been modeled, see Table 1.

System	Motorola DSP56k	ADSP 2106x	TI 320C40
P_L	2	1	1
P_S	2	2	2
T_L, T_L^{init}	0,6	0,1	8,1
T_S	8	2	8

Table 1: The parameters of 3 well-known DSP processors. All are capable of performing zero-overhead looping. For the TMS320C40, however, it is recommended to use a conventional counter and branch implementation of a loop in case of nested loops.

The DSP56k and TMS320C40 have high subroutine execution time overhead; the DSP56k, however, has a zero-loop overhead and high loop initialization overhead; and the TMS320C40 has a high loop iteration overhead but low loop initialization overhead.

4 Evolutionary Multi-objective Optimization

The problem under consideration involves 3 different objectives: program memory, buffer memory, and execution time. These cannot be minimized simultaneously, since they are conflicting – a typical multi-objective optimization problem. In this case, one is not interested in a single solution but rather in a set of optimal trade-offs which consists of all solutions that cannot be improved in one criterion without degradation in another. The corresponding set is denoted as *Pareto-optimal* set.

Evolutionary Algorithms (EAs), a class of probabilistic optimization methods that mimic natural evolution, are especially suited to this kind of problem. They process a *population* of solutions in parallel and sample the search space by means of selection, crossover, and mutation. Therefore, multiple Pareto-optimal solutions can be captured in a single simulation run. Moreover, similarities between different solutions may be exploited using recombination.

Several evolutionary approaches to multi-objective optimization have been proposed since 1985, which mainly differ in the way of fitness assignment. This is the crucial point because EAs operate on scalar fitness values. The question is how to calculate a scalar value from several objectives.

In this study, we use a new technique called *Strength Pareto Evolutionary Algorithm* (SPEA) [10]. It can be summarized as follows: Besides the population a second, external population is maintained which contains the best solutions found so far. In order to determine which solutions are better than others, the concept of Pareto-optimality is applied analogously using the *dominance* relation: A solution a is said to *dominate* (or to be *preferable* to) a solution b iff b is equal to or superior to b in all criteria and at least better in one criterion.[6] Accordingly, all solutions that are not dominated by any other solution found during the evolution process are kept in the external population.

Fitness assignment is performed in 2 stages. In the first stage, the nondominated solutions in the external population are assessed by counting the number of population member they dominate. The less solutions are dominated, the better the fitness of the corresponding nondominated solution. Afterwards, in the second stage, for each element in the population the fitness values of those external nondominated solutions are added up that dominate this particular element. Again, the lower the sum, the better the fitness that is assigned to the individual. Furthermore, it is ensured that all individuals in the population, have worse fitness values than those in the external population. This puts particular emphasis on the nondominated solutions, since individuals are selected from both populations using binary tournament selection with replacement.

The genotype of each solution contains 4 chromosomes. One chromosome represents the schedule, i.e., the order of actor firings. Since for each actor the number of firings are known a priori, the length of the chromosome and the schedule, respectively, is fixed. However, a repair mechanism is necessary to guarantee that every genotype can be unambiguously mapped to a valid schedule; arbitrary actor firing sequences may contain deadlocks, etc. The repair procedure incorporated here bases on a topological sort algorithm and is similar to the one described in [7]. The code model is encoded by a flag, which can be in 3 different states: either all actors are implemented using only subroutine calls or only inlining, or subroutines and inlining are mixed. For the last case, a bit vector is maintained in the genotype which indicates for each actor separately whether it appears as inlined or subroutine code in the implementation. Finally, a 4th chromosome, the loop flag, determines whether to use loops as a mean to reduce program memory. For this aim, a dynamic programming looping procedure is applied to the actor firing sequence in order to find an optimally looped schedule. This procedure, which has been incorporated in our system, is a generalization of the GDPPO procedure presented in [1].

Due to the heterogeneous chromosomes, a mixture of different crossover and mutation operators accomplishes the generation of new individuals. For the schedule, permutation-based operators are necessary that only permutate the actor firing sequence without changing the number of occurrences per actor in the schedule. Here the same operators as in [7] have been used: *order-based uniform* crossover and *scramble sublist* mutation. Concerning the bit vector, we work with the commonly used one-point crossover and bit flip mutation. Similarly, the 2 flags are treated as bit vectors of size 1 and 2, respectively, and therefore processed analogously.

5 Experiments

The CDtoDAT example was taken as the basis to compare the design spaces of 3 different DSP processors. For each processor, the EA ran 100 generations with an overall population size of 350 (the mating pool size was set to 50, the maximum size of the external population equaled 300). The remaining parameters were fixed and chosen

[6]Pareto-optimal points are solutions not dominated by any other solution in the search space.

based on preliminary experiments (crossover probability: 0.8, mutation rate: 0.1). Moreover, a heuristic (APGAN – acyclic pairwise grouping of adjacent nodes [1]) was applied to this problem. The APGAN solution was inserted in 2 ways into the initial population: with and without looping.

The experimental results are visualized in Figures 4, 5, and 6. To make the differences between the processors clearer, the plots have been cut at the top without destroying their characteristics.

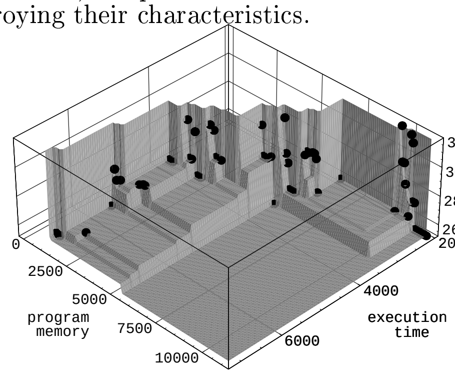

Figure 4: Motorola DSP56k

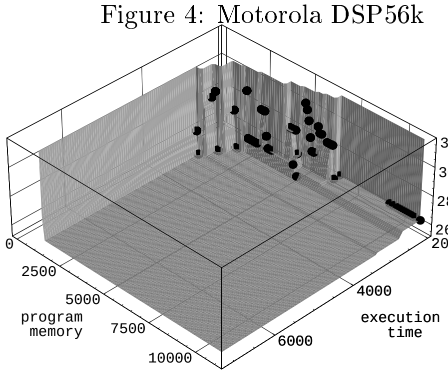

Figure 5: ADSP 2106x

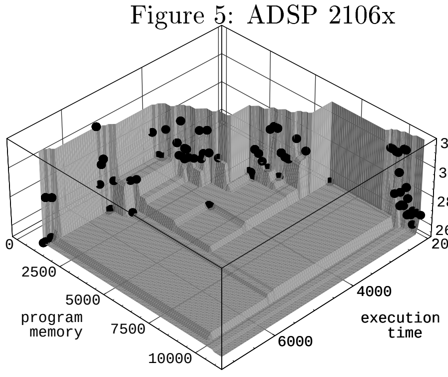

Figure 6: TI TMS320C40

The trade-offs between the 3 objectives are very well reflected by the extreme points. The rightmost points in the plots represent schedules that neither use looping nor subroutine calls. Therefore, there are optimal in the execution time dimension, but need a maximum of program memory because for each actor firing there is an inlined code block. In contrast, the leftmost points make excessive use of looping and subroutines which leads to minimal program memory requirements, however at the expense of a maximum execution time overhead. Another extreme point (not shown in the figures) satisfies $D(S) = 1021$, but has only little overhead in the remaining 2 dimensions. It stands for an implementation which includes the code for each actor only once by using inlining and looping. The schedule associated with this implementation is a single appearance schedule.

Furthermore, the influence of looping and subroutine calls is remarkable. Using subroutines does not interfere with buffer memory requirements; there is only a trade-off between program memory and execution time. Subroutine calls may save much program memory, but at the same time they are expensive in terms of execution time. This fact is reflected by "gaps" on the execution time axis in Figure 4 and 6. Looping, however, depends on the schedule: schedules which can be looped well may have high buffer memory requirements and vice versa. This trade-off is responsible for the variations in buffer memory requirements and is illustrated by the points that are close to each other regarding program memory and execution time, but strongly differ in the buffer memory required. Comparing the figures, one can observe that the ADSP 2106x produces less execution time overhead than the other processors which is in accordancewith Table 1. Subroutine calls are most frequently used in case of the TMS320C40 because of the high loop iteration overhead.

References

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

[2] S.S. Bhattacharyya, P.K. Murthy, and E.A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *invited paper, J. of VLSI Signal Processing*, page to appear, 1999.

[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] 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.

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

[6] 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.

[7] J. Teich, E. Zitzler, and S. S. Bhattacharyya. Buffer memory optimization in dsp applications — an evolutionary approach. In *Fifth International Conference on Parallel Problem Solving from Nature (PPSN-V)*, pages 885–894, 1998.

[8] J. Teich, E. Zitzler, and S. S. Bhattacharyya. 3d exploration of uniprocessor schedules for dsp algorithms. Technical Report 56, Institute TIK, ETH Zurich, Switzerland, Jan 1999.

[9] P. P. Vaidyanathan. *Multirate Systems and Filter Banks.* Prentice Hall, 1993.

[10] E. Zitzler and L. Thiele. An evolutionary algorithm for multiobjective optimization: The strength pareto approach. Technical Report 43, Institute TIK, ETH Zurich, Switzerland, May 1998.

[11] V. Zivojnovic, J. Martinez, C. Schläger, and H. Meyr. A DSP-oriented benchmarking methodology. In *Int. Conf. on Sig. Proc: Applications & Technology*, 1994.