Evolutionary Algorithm Based Exploration of Software Schedules for Digital Signal Processors

Eckart Zitzler
EE Dept. and Institute TIK
ETH Zürich, Gloriastr. 35
CH–8092 Zürich, Switzerland
zitzler@tik.ee.ethz.ch

Jürgen Teich
EE Dept. and Institute DATE
University of Paderborn
D–33098 Paderborn, Germany
teich@date.uni-paderborn.de

Shuvra S. Bhattacharyya
EE Dept. and UMIACS
University of Maryland
College Park, MD 20742, U.S.A.
ssb@eng.umd.edu

Abstract

The simultaneous exploration of tradeoffs between *program memory*, *data memory* and *execution time* requirements (3D) for DSP (digital signal processing) algorithms in embedded computing environments is a demanding application and example par excellence of a *multi-objective optimization* problem. In order to solve this problem, two *evolutionary algorithms* are shown to be successfully applicable for exploring *Pareto-optimal* solutions. For different well-known target DSP processors, the trade-off fronts are analyzed. The two approaches are quantitatively compared.

1 Introduction

Starting with a data flow graph specification to be implemented on a digital signal processor, we study the effects between instantiating code 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 environment, e.g., a single-chip solution. For such applications, a careful exploration of the possible spectrum of implementations is of utmost importance because the market of these products is driven by tight cost and performance constraints. Frequently, these systems are once programmed to run forever. Optimization and exploration times in the order of hours are therefore neglectable.

We present the first systematic optimization framework for exploring implementation trade-offs in the 3-dimensional run-time/program memory/data memory space of implementations, and we compare two evolutionary algorithm based Pareto-front exploration approaches to solve this multi-objective optimization problem.

The methodology begins with a given *synchronous dataflow graph* [Lee and Messerschmitt, 1987] as used in many rapid prototyping environments as input for code generators for programmable digital signal processors (PDSPs) [Buck et al., 1991, Lauwereins et al., 1990, Ritz et al., 1992].

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 [Vaidyanathan, 1993] for details on multistage sample rate conversion.*

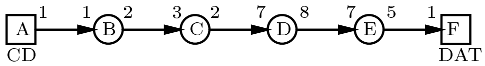

Figure 1: CDtoDAT conversion benchmark

As reported by DSP analysts (e.g., the DSPStone benchmarking group [Zivojnovic et al., 1994]), 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 [Bhattacharyya et al., 1996]. 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 [Bhattacharyya et al., 1999] with inline code) would be preferable.

The present study extends our previous work [Teich et al., 1998] where a single-objective EA was used to minimize data memory requirements for a restricted class of schedules (SAS) and implementations (no subroutine calls). Here, we seek to explore the dimensions of program memory, data memory, and execution time requirements simultaneously for arbitrary schedules, a demanding multi-objective optimization problem. For its solution, two evolutionary algorithm based approaches are compared in Section 4 First, the optimization problem and metrics are formally defined (Section 2 and 3). Sections 5 and 6 deal with aspects of the EAs, and the experiments performed, respectively, including a quantitative comparison of the two EAs for solving the exploration problem.

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 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 $produced(\alpha)$ 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 $delay(\alpha)$ that reside on α.*

Example 2 *The graph in Fig. 1 has $|V| = 6$ nodes (or actors). Each presents a function that may be executed as soon as its input contains at least $consumed(\alpha)$ data tokens on each ingoing arc α, see the numbers annotated with the arc heads. E.g., actor B requires one input token on its input arc, and produces 2 output tokens on its outgoing arc when firing. In the shown graph, $delay(\alpha) = 0\ \forall \alpha \in A$. Hence, initially, only actor A, the source node, may fire. Afterwards, B may fire for the first time. After that, however, node C still cannot yet fire, because it requires $consumed(\alpha) = 3$ tokens on its ingoing arc, however, there are only two produced by the firing of B. In general, many firing sequences of actors may evolve.*

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 3 *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 4 *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 5 *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 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.*

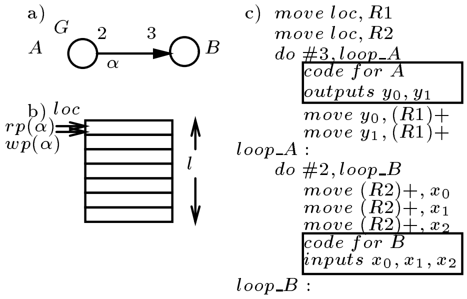

Figure 2: SDF graph a), memory model for arc buffer b), and Motorola DSP56k-like assembly code realizing the schedule $S = (\infty(3A)(2B))$.

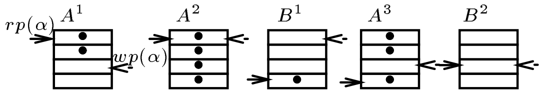

Figure 3: Memory accesses for the schedule $S = (\infty(3A)(2B))$

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 occurrence 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]

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

Note that 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 increment, loop exit testing and branching instructions. This overhead is processor-specific, and in our computations proportional to the number of loops in the schedules.

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

3.2 Buffer memory overhead $D(S)$

We account for overhead due to data buffering for the communication of actors (buffer cost). 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:

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

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

Example 6 *Consider the schedule in Example 4 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]

[2] In [Teich et al., 1999], 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.

[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

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

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

where T_{io} denotes the number of clock cycles that are needed between reading (writing) 2 successive 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

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

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

Example 7 *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

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

Example 8 *Consider again Example 7. 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).
- 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 and one fictive processor P1 have been modeled, see Table 1. One can observe that the DSP56k and TMS320C40 have high subroutine execution time overhead; the DSP56k, however, has a zero-loop overhead and high loop initialization overhead;

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.P1 is a fictive processor modeling high subroutine overheads.

System	Motorola DSP56k	ADSP 2106x	TI 320C40	P1
P_L	2	1	1	2
P_S	2	2	2	10
T_L, T_L^{init}	0,6	0,1	8,1	0,1
T_S	8	2	8	16

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.

and the TMS320C40 has a high loop iteration overhead but low loop initialization overhead. P1 models a processor with high subroutine overheads.

4 Evolutionary Multi-objective Optimization

The problem under consideration involves three 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.

Definition 2 *Let us consider, without loss of generality, a multi-objective minimization problem with m decision variables and n objectives:*

$$\textit{Minimize} \quad \vec{y} = f(\vec{x}) = (f_1(\vec{x}), \ldots, f_n(\vec{x})) \tag{6}$$

where $\vec{x} = (x_1, \ldots, x_m) \in X$ and $\vec{y} = (y_1, \ldots, y_n) \in Y$ are tuples. A decision vector $\vec{a} \in X$ is said to dominate *a decision vector $\vec{b} \in X$ (also written as $\vec{a} \succ \vec{b}$) iff*

$$\begin{aligned} \forall i \in \{1, \ldots, n\}: \quad & f_i(\vec{a}) \leq f_i(\vec{b}) \qquad \wedge \\ \exists j \in \{1, \ldots, n\}: \quad & f_j(\vec{a}) < f_j(\vec{b}) \end{aligned} \tag{7}$$

Additionally, in this study we say $\vec{a}$ covers *$\vec{b}$ ($\vec{a} \succeq \vec{b}$) iff $\vec{a} \succ \vec{b}$ or $f(\vec{a}) = f(\vec{b})$. All decision vectors which are not dominated by any other decision vector are called* nondominated. Pareto-optimal *points are the nondominated decision vectors of the entire search space.*

Evolutionary algorithms (EAs) seem to be especially suited to multi-objective optimization because they are able to capture multiple Pareto-optimal solutions in a single simulation run and may exploit similarities of solutions by crossover. Some researchers even suggest that multi-objective search and optimization might be a problem area where EAs do better than other blind search strategies [Fonseca and Fleming, 1995]. The fact that several multi-objective EAs have been proposed since 1985[6] and that the interest in that field has been growing up to now supports this assumption.

In this study, the Strength Pareto Evolutionary Algorithm (SPEA), a recent technique proposed in [Zitzler and Thiele, 1998a], is used. In Section 6.2, it is compared with another approach called Niched Pareto Genetic Algorithm (NPGA) [Horn and Nafpliotis, 1993]. Both methods are briefly described in the following.

[6] An excellent review of different evolutionary approaches can be found in [Fonseca and Fleming, 1995].

4.1 Strength Pareto Evolutionary Algorithm

SPEA maintains besides the regular population an external set of individuals that contains the nondominated solutions of all solutions generated so far. This set is updated every generation and if necessary reduced in size in case the maximum number of members is exceeded. The reduction is accomplished by a clustering technique which preserves the characteristics of the nondominated front.

Fitness assignment is performed in two steps:

Step 1: Each solution i in the external nondominated set is assigned a real fitness value $f_i \in [0, 1)$, where f_i is the number of population members j, for which $i \preceq j$, divided by the population size plus one.

Step 2: The fitness of an individual j in the population is calculated by summing the fitness values of all external nondominated solutions i that cover j.[7]

Finally, both population and external nondominated set take part in the selection process. Thereby, binary tournament selection with replacement is used to fill the mating pool.

4.2 Niched Pareto Genetic Algorithm

NPGA combines tournament selection and the concept of Pareto dominance. Two competing individuals and a comparison set of t_{dom} other individuals are picked at random from the population. If one of the competing individuals is dominated by any member of the set and the other is not, then the latter is chosen as winner of the tournament. If *both* individuals are dominated (or not dominated), the result of the tournament is decided by *fitness sharing* (see, e.g., [Deb and Goldberg, 1989]): The individual which has less individuals in its niche (defined by the parameter σ_{share}) is selected for reproduction.

5 Problem Coding

Each genotype consists of four parts which are encoded in separate chromosomes:

1. schedule,
2. code model,
3. actor implementation vector, and
4. loop flag.

The schedule represents the order of actor firings and is fixed in length because the number of firings of each actor is known a priori. Since arbitrary actor firing sequences may contain deadlocks, etc., a repair mechanism is applied in order to map every schedule chromosome unambiguously to a valid schedule. It

[7] Since small fitness values correspond to high reproduction probabilities, members of the external nondominated set have better fitness than the population members.

bases on a topological sort algorithm, and is described in [Teich et al., 1998]: at each point in time, the left-most fireable actor is chosen whose maximum number of firings has not been reached yet.

The code model chromosome determines the way how the actors are implemented and contains one gene with three possible alleles: all actors are implemented as subroutines, only inlining is used, or subroutines and inlining are mixed. For the last case, the actor implementation vector, a bit vector, encodes for each actor separately whether it appears as inlined or subroutine code in the implementation.

Finally, a fourth chromosome, the loop flag, determines whether to use loops as a mean to reduce program memory. For this aim, a dynamic programming *looping algorithm* 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 algorithm presented in [Bhattacharyya et al., 1996]. Since the run-time is rather high ($\mathcal{O}(n^3)$) considering large n, the algorithm can be sped up by certain parameter settings—however, at the expense of optimality.[8]

Due to the heterogeneous chromosomes, a mixture of different crossover and mutation operators accomplishes the generation of new individuals. For the schedule, *order-based uniform* crossover and *scramble sublist* mutation are used [Davis, 1991]. These operators only permute the actor firing sequence and guarantee that the number of occurrences per actor remains constant. Concerning the other chromosomes, we work with one-point crossover and bit flip mutation (as the code model gene is represented by an integer, mutation is done by choosing one of the three alleles at random).

6 Experiments

Two kinds of experiments were performed: design space exploration for the CDtoDAT example using different processors and comparison of SPEA and NPGA on nine practical examples. In all cases, the following EA parameters were used:

generations	:	250
population size	:	100
crossover rate	:	0.8
mutation rate	:	0.1[9]

Moreover, before every run a heuristic called APGAN (acyclic pairwise grouping of adjacent nodes [Bhattacharyya et al., 1996]) was applied to this problem. The APGAN solution was inserted in two ways into the initial population: with and without looping.

[8]This extension is called *suboptimal looping* in the following.

[9]The bit vector was mutated with a probability of $1/L$ per bit, where L denotes the length of the vector.

Finally, the set of all nondominated solutions found during the entire evolution process was considered as outcome of one single optimization run.

6.1 CDtoDAT Design Space Exploration

SPEA was used to compare the design spaces of the different DSP processors listed in Table 1; thereby, the size of the external nondominated set was unrestricted in order to find as many solutions as possible.

The experimental results are visualized in Fig. 4 to 8. Four times, the accelerated looping algorithm has been used (about 5 hours run-time on a Sun ULTRA 30), one run has also been made with optimal looping (run-time about 5 days).[10]

To make the differences between the processors clearer, the plots have been cut at the top without destroying their characteristics.

The trade-offs between the three 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, they 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.

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 Figures 5 to 7. Looping, however, depends on the schedule: schedules which

[10]Note that this optimization time is still quite low for processor targets assumed to be programmed once and supposed to run an application forever.

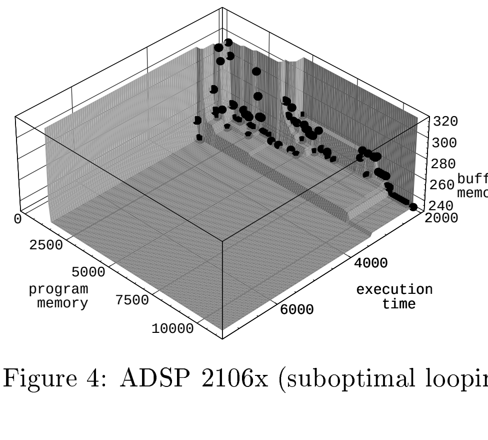

Figure 4: ADSP 2106x (suboptimal looping)

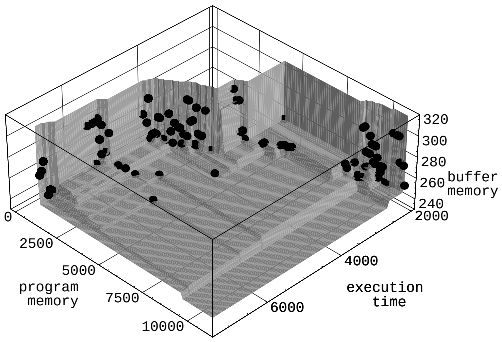

Figure 5: TI TMS320C40 (suboptimal looping)

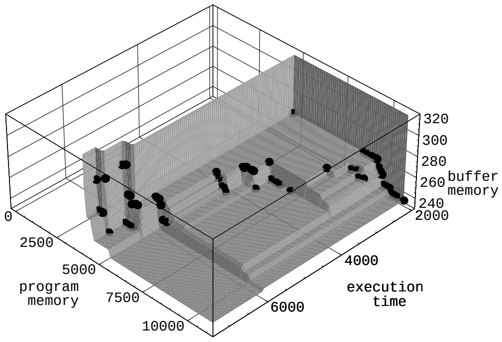

Figure 6: Processor P1 (suboptimal looping)

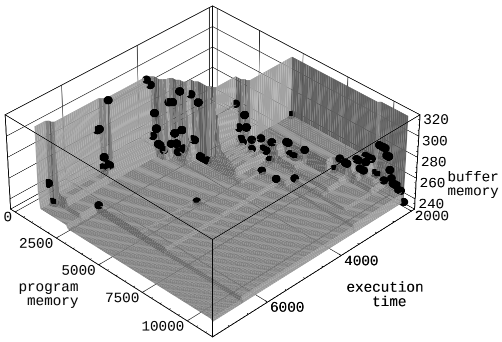

Figure 7: Motorola DSP56k (suboptimal looping)

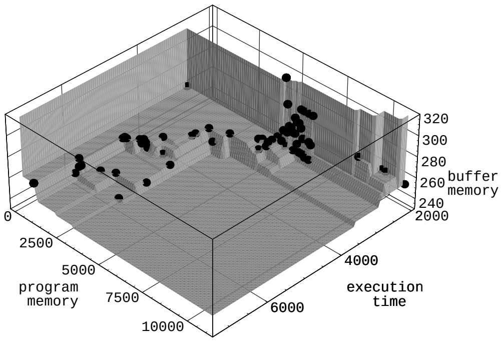

Figure 8: Motorola DSP56k (optimal looping)

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 three real processors regarding suboptimal looping, one can observe that the ADSP 2106x produces less execution time overhead than the other processors which is in accordance with Table 1. Subroutine calls are most frequently used in case of the TMS320C40 because of the high loop iteration overhead.

For processor P1 (Fig. 6), it can be seen that points at the front of minimal program memory require much more program memory than the other processors. This is in accordance with the high penalty in program memory and execution time when subroutines are used. In fact, none of the 186 nondominated points found used subroutine calls for any actor.

The effect of the looping algorithm on the obtained nondominated front can be clearly seen by comparing Figs. 7 and 8: Much buffer memory may be saved in case the optimal looping algorithm is used, the trade-off surface becoming much more flat in this dimension. It is also remarkable that a point with program memory requirements of 171 was found lowest as opposed to 961 for a point with lowest program memory requirements when using the suboptimal looping algorithm. As a result, the optimization time spent by the looping algorithm has a big influence on the shape of the nondominated front.

6.2 Comparison of SPEA and NPGA

Nine practical DSP applications, which were taken from [Bhattacharyya et al., 1996], form the basis to compare the performance of SPEA and NPGA. The number of actors of the corresponding SDF graphs varies between 12 and 92, the length of the associated schedules ranges from 30 to 313 actor firings.

To evaluate the performance of the two EAs, a metric introduced in [Zitzler and Thiele, 1998b] is used here:

Definition 3 *Let A and B be two sets of decision vectors. The function $\mathcal{C}$ maps the ordered pair (A,B) to the interval [0,1]:*

$$\mathcal{C}(A, B) := \frac{|\{b \in B; \exists\, a \in A : a \succeq b\}|}{|B|} \qquad (8)$$

The value $\mathcal{C}(A,B)$ gives the fraction of B that is covered by members of A. Note that both $\mathcal{C}(A,B)$ and $\mathcal{C}(B,A)$ have to be taken into account, since not necessarily $\mathcal{C}(A,B) = 1 - \mathcal{C}(B,A)$. Although, this metric does not say anything about the distributions and the distances of the two fronts, it is sufficient here, as the results will show.

On each example, SPEA and NPGA ran in pairs on the same initial population, using optimal looping; then the two resulting nondominated sets were assessed by means of the $\mathcal{C}$ function. Altogether, eight of these pairwise runs were performed per application, each time operating on a different initial population. Furthermore, in the SPEA implementation the population size was set to 80 and the size of the external nondominated set was limited to 20. Concerning NPGA, we followed recommendations given in [Horn and Nafpliotis, 1993] and chose $t_{\text{dom}} = 10$ (10% of the population size); the niching parameter $\sigma_{\text{share}} = 0.4886$ was calculated based on guidelines given in [Deb and Goldberg, 1989].

Table 2: Comparison of SPEA and NPGA on nine practical examples[11]

System	$\mathcal{C}$(SPEA,NPGA)			$\mathcal{C}$(NPGA,SPEA)		
	mean	min	max	mean	min	max
1	99%	97%	100%	12%	10%	17%
2	98%	93%	100%	34%	14%	53%
3	94%	78%	100%	6%	5%	8%
4	99%	95%	100%	4%	2%	8%
5	99%	96%	100%	4%	2%	10%
6	91%	79%	98%	7%	4%	10%
7	97%	89%	100%	8%	5%	17%
8	98%	93%	100%	3%	2%	3%
9	97%	90%	100%	4%	2%	7%

The experimental results are summarized in Table 2. On all nine applications, SPEA covers more than 78% of the NPGA outcomes (in average more than 90%), whereas NPGA covers in average less than 10% of the SPEA outcomes. This means that the fronts generated by SPEA dominate most parts of the corresponding NPGA fronts, whereas only very few solutions found by NPGA are not covered. Since SPEA incorporates an elitist strategy in contrast with NPGA, the results suggest that elitism might be an important factor to improve evolutionary multi-objective search. This observation was also made in [Zitzler and Thiele, 1998a].

References

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

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

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

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

[Deb and Goldberg, 1989] Deb, K. and Goldberg, D. E. (1989). An investigation of niche and species formation in genetic function optimization. In Schaffer, J. D., editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 42–50. Morgan Kaufmann.

[Fonseca and Fleming, 1995] Fonseca, C. M. and Fleming, P. J. (1995). An overview of evolutionary algorithms in multiobjective optimization. *Evolutionary Computation*, 3(1):1–16.

[Horn and Nafpliotis, 1993] Horn, J. and Nafpliotis, N. (1993). Multiobjective optimization using the niched pareto genetic algorithm. IlliGAL Report 93005, Illinois Genetic Algorithms Laboratory, University of Illinois, Urbana, Champaign.

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

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

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

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

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

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

[Zitzler and Thiele, 1998a] Zitzler, E. and Thiele, L. (1998a). An evolutionary algorithm for multiobjective optimization: The strength pareto approach. Technical Report 43, Institute TIK, ETH Zurich, Switzerland. Submitted to IEEE Transactions on Evolutionary Computation.

[Zitzler and Thiele, 1998b] Zitzler, E. and Thiele, L. (1998b). Multiobjective optimization using evolutionary algorithms — a comparative case study. In *Fifth International Conference on Parallel Problem Solving from Nature (PPSN-V)*, pages 292–301.

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

[11] The following systems have been considered: 1) fractional decimation; 2) Laplacian pyramid; 3) nonuniform filterbank (1/3, 2/3 splits, 4 channels); 4) QMF nonuniform-tree filterbank; 5) QMF filterbank (one-sided tree); 6) QMF analysis only; 7) QMF tree filterbank (4 channels); 8) QMF tree filterbank (8 channels); 9) QMF tree filterbank (16 channels), see [Bhattacharyya et al., 1996] for details.