Multidimensional Exploration of Software Implementations for DSP Algorithms *

Eckart Zitzler (zitzler@tik.ee.ethz.ch)
Computer Engineering and Networks Laboratory (TIK), ETH Zurich, Switzerland

Jürgen Teich (teich@date.uni-paderborn.de)
Computer Engineering (DATE), University of Paderborn, Germany

Shuvra S. Bhattacharyya (ssb@eng.umd.edu)
Institute for Advanced Computer Studies (UMIACS), University of Maryland, College Park, USA

July 1, 1999

Abstract. When implementing software for programmable digital signal processors (PDSPs), the design space is defined by a complex range of constraints and optimization objectives. Three implementation metrics that are crucial in many PDSP applications are the *program memory requirement (code size)*, *data memory requirement*, and *execution time*. This paper addresses the problem of exploring the 3-dimensional space of trade-offs that is defined by these crucial metrics. Given a software library for a target PDSP, and a dataflow-based block diagram specification of a DSP application in terms of this library, our objective in this paper is to compute a full range of *Pareto-optimal solutions*. For solving this multi-objective optimization problem, an *evolutionary algorithm* based approach is applied. We illustrate our techniques by analyzing the trade-off fronts of a practical application for a number of well-known, commercial PDSPs.

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 (PDSP) 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 tolerable.

* S. S. Bhattacharyya was supported in this work by the US National Science Foundation (CAREER, MIP9734275) and Northrop Grumman Corp.

We present the first systematic optimization framework for exploring trade-offs in the space of possible software implementations with regard to the following three objectives: execution time, program memory, and data memory.

The methodology begins with a given *synchronous dataflow graph* [12], a restricted form of dataflow in which the nodes, called *actors* have a simple firing rule: The number of data values (*tokens, samples*) produced and consumed by each actor is fixed and known at compile-time.

EXAMPLE 1. *A practical example is a sample-rate conversion system. In Figure 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 [15] for details on multistage sample rate conversion.*

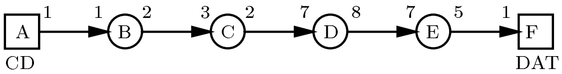

Figure 1. CDtoDAT conversion benchmark

The major reason why the SDF model is widely used as the underlying specification model are the abilities to express multirate systems, parallelism, and that many important aspects such as deadlock detection and scheduling can be determined at compile-time.

As a matter of fact, there exist rapid prototyping environments that use SDF graphs or related models as input for code generators for programmable digital signal processors (PDSPs) [5, 11, 13].

As reported by DSP analysts (e.g., the DSPStone benchmarking group [18]), today's DSP compilers still produce several 100%s of overhead with respect to assembly code written and optimized by hand. Hence, the hardware capabilities such as *zero-loop overhead*, execution of multiple instructions (e.g., memory write and execute), and specialized instruction sets (e.g., multiply-add operations) as well as special addressing modes (e.g., indexed versus pointer addressing) cannot be sufficiently exploited at this point in time.

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 (.e.g., filter components) 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.

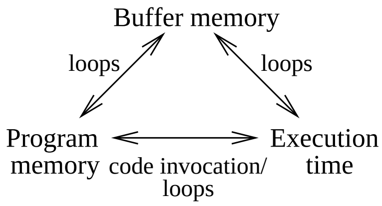

Figure 2. Trade-off between the three performance criteria

Prior work on code size minimization of SDF schedules has focused on an inline code generation model [4]. 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.

Figure 2 depicts the trade-off that has to be made. The buffer memory is mainly influenced by the schedule and the chosen buffer model. Software loops increase the execution time, however, decrease the program memory. Since it depends on the schedule to which extent looping can be applied, there is a trade-off between program memory and data memory. The use of subroutine calls causes a higher execution time, but decreases again the program memory requirements. With inlining, the target code gets faster, but larger.

Because of this trade-off, the software synthesis based on a SDF specification is a typical multi-objective optimization problem. Three conflicting goals have to be optimized, which are all of equal importance. This means, that many "best" solutions coexist. The optimization process should find the set of this "best" solutions.

One approach to perform this exploration are *evolutionary algorithms* (EAs). These are optimization methods based on nature's evolution process. Solutions in the search space are considered as individuals of a population. A selection mechanism will give preference to strong individuals and disfavor weaker ones similar to the survival-of-the-fittest principle. Figure 3 shows the basic mechanisms of an EA. The iterations represent the sequence of generations of the population.

This paper is organized as follows. First, the optimization problem and metrics are formally defined (Section 2 and 3). A code-size dynamic programming post-optimization algorithm called CDPPO which is implemented in the evolutionary framework for computing optimally

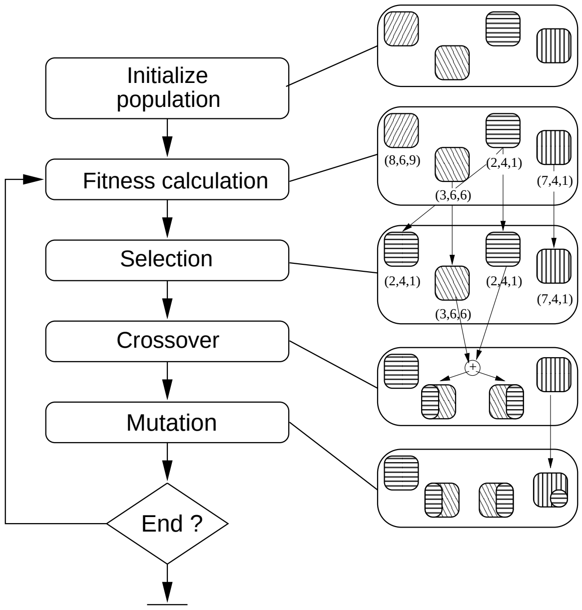

Figure 3. Outline of an evolutionary algorithm

nested loops for a given schedule, is described in Section 4. Afterwards, basic principles of multi-objective optimization are briefly discussed in Section 6, and Section 7 deals with implementation aspects of the EA. Finally, Section 8 presents the experimental results concerning the CDtoDAT application (cf. Example 1), and the last section is devoted to concluding remarks and future perspectives.

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 : $A \rightarrow \mathbf{N}_0$ *denotes the function that assigns to each arc* $\alpha \in A$ *the number of initial tokens delay*(α) *that reside on* α.

EXAMPLE 2. *The graph in Figure 1 has* $|V| = 6$ *nodes (or actors). Each presents a function that may be executed as soon as its input contains at least consumed*(α) *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*.

SDF graphs for which valid schedules exist are called *consistent* graphs. Systematic techniques exist to efficiently determine whether or not a given SDF graph is consistent and to compute the minimum number of times that each actor must execute in the body of a valid schedule [12]. We represent these minimum numbers of firings by a function q_{G} or simply q in case G is known from the context with $q : V \rightarrow \mathbf{N}$.

EXAMPLE 3. *The CDtoDAT graph in Figure 1 is consistent because there exists a (non-zero) finite actor firing sequence such that the initial token configuration is obtained again. 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 following schedule is a valid SAS for the graph shown in Figure 1:* $(\infty(147A)(147B)(98C)(28D)(32E)(160F))$.

Although SAS offer minimal code size, they are not necessarily optimal in terms of data-memory consumption [2]. In general, a schedule of the form $(\infty\ (q(N_1)N_1)\ (q(N_2)N_2)\ \cdots\ (q(N_{|\mathrm{V}|})N_{|\mathrm{V}|}))$ where N_i denotes the (label of the) ith node of a given SDF graph, and |V| denotes the number of nodes of the given graph, is called *flat single appearance schedule.*

EXAMPLE 5. *Consider the simple SDF graph G in Figure 5a). With $produced(\alpha) = 2$ and $consumed(\alpha) = 3$, we obtain $q(A) = 3$, $q(B) = 2$ as the minimal actor repetition numbers. The schedule $(\infty(3A)(2B))$ is a valid (flat) SAS, requiring* 6 *units of memory to store the maximal amount of data that accumulates on the arc α (after the firing of the first schedule loop). The schedule $(\infty(2A)BAB)$ requires only 4 units (after the first two firings of actor A).*

The above example shows us that SASs, though compact, are not necessarily data-memory minimal. In order to exploit also multiple-appearance schedules (called MAS in the following), we must be able to look at the *unfolding* of the SDF graph that represents the causal properties of each single actor invocation in a periodic schedule.

DEFINITION 2. (Unfolding). *Given a consistent SDF graph $G = (V, A, consumed, produced, delay)$ and the minimal repetition vector $q = (q(N_1), q(N_2), \ldots, q(N_{|V|}))$. The marked graph $G' = (V', A', delay')$ obtained as follows is called its* unfolding*:*

$-|V'| = \sum_{i=1}^{|V|} q(N_i)$ and for each node $N_i \in V$ there exists a set $V_i' = \{N_i^1, \ldots, N_i^{q(N_i)}\}$ in V' ($V' = \bigcup_{i=1}^{|V|} V_i'$).

$-$The arc set A' and the initial data $delay'$ are obtained as follows (we consider each arc $\alpha = (N_i, N_j) \in A$ separately and instantiate corresponding arcs in A'). Thereby, δ_i, with $1 \leq \delta_i \leq q(N_i)$, denotes the counting index for the $q(N_i)$ instances $N_i^{\delta_i}$.

For each arc $\alpha = (N_i, N_j) \in A$, there exist $q(N_i) \cdot produced(\alpha)$ arcs $\alpha' = (N_i^{\delta_i}, N_j^{\delta_j}) \in A'$, where the kth arc with $k = 1, \ldots, q(N_i) \cdot produced(v_i, v_j)$) is directed from node $N_i^{\delta_i}$ with

$$\delta_i = ((k-1) \operatorname{div} produced(N_i, N_j)) + 1$$

to node $N_j^{\delta_j}$ with

$$\delta_j = (((delay(\alpha) + k - 1) \bmod (q(N_j) consumed(\alpha))) \operatorname{div} consumed(\alpha)) + 1.$$

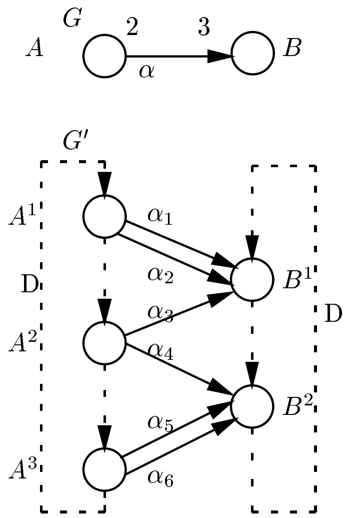

Figure 4. Unfolding of the SDF graph as introduced in Example 5.

Furthermore, the lth initial token with $l = 1, \ldots, delay(\alpha)$ is situated on the kth arc between nodes N_i and N_j in A' with

$$k = q(N_i)produced(\alpha) - (l - 1) \bmod (q(N_i)produced(\alpha))$$

Obviously, the unfolding $G' = (V', A', delay')$ contains

- $|V'| = \sum_{i=1}^{|V|} q(N_i)$ nodes, and
- $|A'| = \sum_{j=1}^{|A|} produced(\alpha_j) \cdot consumed(\alpha_j)$ arcs.

EXAMPLE 6. *Consider again the simple SDF graph G as introduced in Example 5. The unfolding of this graph is shown in Figure 4. There are three actor instantiations of actor A and two instances of actor B. It can be shown that single-appearance schedules of this graph represent all valid schedules of the corresponding SDF graph. For example, the schedule $(\infty AABAB)$ is a valid schedule, whereas $(\infty ABABA)$ is not. Vice versa, if a schedule of the unfolding is not valid, then the schedule of actor names is also invalid in the original SDF graph. In this sense, both of them are equivalent.*

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

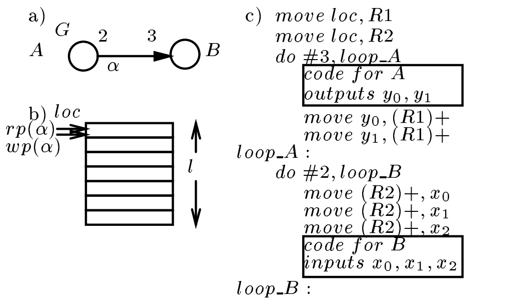

Figure 5. SDF graph a), memory model for arc buffer b), and Motorola DSP56k-like assembly code realizing the schedule $S = (\infty(3A)(2B))$

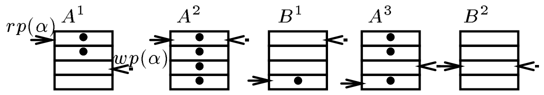

Figure 6. Memory accesses for the schedule $S = (\infty AABAB)$

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 7. *For the simple SDF graph in Figure 5a), 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 Figure 5b), 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) are shown in Figure 6.*

Thus, for a valid schedule S, the code for each actor has to be instantiated from a predefined library corresponding to the sequence of actor firings of S. This can happen in two ways: the code can either be inserted in line into the target code or be invoked as a subroutine call. The mode of code invocation should be specified for each actor separately. We introduce a function $flag(N_i) \in \{0, 1\}$ that indicates for each actor N_i whether it should be instantiated by a subroutine call ($flag(N_i) = 0$) or inlined into the final program code ($flag(N_i) = 1$) for all actor invocations.

Also, it can also be specified whether software loops should be implemented or not. If loops are desired, each schedule loop is translated into a software loop.

In the end, the whole resulting code block is encapsulated within an infinite loop to obtain a software implementation of the periodic schedule.

EXAMPLE 8. (A schedule and different implementations). *Considering the schedule $AAABB$ of the SDF graph depicted in Figure 5a), the figure below shows some possible software implementations in assembly code. The first uses code inlining only. The second invokes the code with subroutine calls. The last implementation uses program loops corresponding to the single appearance schedule* $(3A)(3B)$.

```
code block A
 ...
code block A
 ...
code block A
 ...
code block B
 ...
code block B
 ...
```

Inlining

```
A
 code block A
  ...
B
 code block B
  ...
JSR A
JSR A
JSR A
JSR B
JSR B
```

Subroutine calls

```
do #3, loopA
 code block A
  ...
loopA

do #2, loop B
 code block B
  ...
loopB
```

Program loops

Figure 7. Different ways of generating code using inlining (left), subroutine calls (middle), and loops with inlined code. Many intermediate solutions, e.g., using loops and subroutine calls, may be imagined.

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]

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

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. Let P_L denote the processor specific constant that models the number of code words for an entire loop instruction. We have to add this constant as many times as schedule loops appear in a schedule S. Assume that $n_{loop}(S)$ is the total number of loops in the schedule S. Thus, the loop overhead $PL(S)$ becomes:

$$P_L(S) = n_{loop}(S) \cdot P_L \tag{2}$$

3.2. Buffer memory overhead $D(S)$

Note that the individual actor code blocks might need data memory for storing local variables (e.g., filter coefficients, counter variables, etc.). As this is a fixed amount of overhead for each actor, and as this amount is independent from the fact whether inlining or subroutine calls are used, we account only for overhead due to data buffering for the communication of actors (buffer cost).

There are many different ways the buffer memory can be organized. The memory can be allocated fixed to one actor or it can be shared during execution of a schedule. A buffer block can be considered as existing the whole schedule period or only as long as it is used. Obviously, sharing of buffer will in general reduce memory requirements. On the

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

other hand a more complex buffer organization will cause an additional overhead for storing information about the structure of the memory.

For simplicity, it is assumed here that a distinct segment of memory is allocated for each arc of a given graph. For this unshared buffer model, the amount of data needed to store the tokens that accumulate on each arc during the evolution of a schedule S is given as:

$$D(S) = \sum_{\alpha \in A} max\_tokens(\alpha, S) \tag{3}$$

Here, $max\_tokens(\alpha, S)$ denotes the maximum number of tokens that accumulate on arc α during the execution of schedule S. This is the minimal size required for one buffer and can easily be computed by simulating the schedule execution and tracing the tokens accumulating on the arc α.

EXAMPLE 9. *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, for the looped schedule* $(\infty(7(7(3AB)(2C))(4D))(32E(5F)))$ *the buffer memory requirement 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.[2]

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 for which $flag(N_i) = 0$, 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).[3] For a given schedule S, we define the

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

[3] 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 al-

subroutine overhead $SO(S)$ as:

$$SO(S) = \sum_{i=1}^{|V|} (1 - flag(N_i)) \cdot L(N_i) \cdot q(N_i) \tag{4}$$

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.[4], 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

$$IO(S) = 2 \sum_{\alpha=(N_i,N_j)\in A} q(N_i) produced(N_i) T_{io} \tag{5}$$

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

$$O(L) = T_L^{init} + x(L) \cdot \left(T_L + \sum_{L' \text{ evoked in} L} O(L') \right) \tag{6}$$

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

lowed. 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).

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

EXAMPLE 10. *Consider an SDF graph with 4 actors* $\{A, B, C, D\}$, *and furthermore let* $(\infty(3(3A)(4B))\ (4(3C)(2D)))$ *be a valid schedule for this graph. 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

$$T(S) = SO(S)\ +\ IO(S)\ +\ O(S) \tag{7}$$

EXAMPLE 11. *Consider again Example 10. 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.

Table I. 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

Three real PDSPs and one fictive processor P1 have been modeled, see Table I. 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; and the TMS320C40 has a high loop iteration overhead but low loop initialization overhead. P1 models a processor with high subroutine overheads.

4. CDPPO: Code-size dynamic programming post-optimization

Prior work on the construction of looped schedules for SDF graphs has focused on single appearance schedules. A single appearance schedule can be uniquely represented as a *parenthesization* of a *lexical ordering* of the input SDF graph, and vice versa [4]. Here, by a lexical ordering, we simply mean a linear ordering $(a_1, a_2, \ldots, a_N)$ of the actors in the graph. Different parenthesizations of a given lexical ordering lead in general to different schedules (sequences of actor invocations), and thus, to different buffer memory requirements.

The problem of optimally parenthesizing a single appearance schedule to minimize the buffer memory requirement has been shown to have a similar structure as the *matrix chain multiplication problem* [6], which is the problem of determining the most efficient order (sequence of pairwise matrix multiplications) in which to evaluate a product $M_1 M_2 \ldots M_k$ of matrices. It is well-known that the matrix-chain multiplication problem is amenable to an efficient dynamic programming solution [6]. The GDPPO algorithm presented in [4] can be viewed as an adaptation of this dynamic programming solution to the parenthesization problem for single appearance schedules.

As discussed in Section 1, our multi-objective optimization techniques developed in this paper go beyond the class of single appearance schedules since multiple appearance schedules can provide significantly lower buffer memory requirements over the best single appearance schedules: when attempting to determine the Pareto-optimal front of an SDF design space, it is essential to consider multiple appearance schedules. When considering arbitrary schedules (schedules that are not necessarily single appearance schedules), a new parenthesization problem becomes relevant. This is the problem of parenthesizing groups of repetitive invocation patterns that can be consolidated into loops. Formally, this is the problem of computing a minimum code size loop hierarchy for a given sequence of actor invocations, and a given code size $C(i)$ associated with each actor invocation i. Typically, all invocations of an actor A that are inlined will have the same code size cost $C_i(A)$, and all invocations that are executed with subroutine calls will have the code size cost of the associated call $C_s(A)$. In such cases, an actor A that has both inlined and subroutine call invocations in the given invocation sequence can be handled by decomposing the actor into two separate actors (within the invocation sequence) — one actor A_i that represents the inlined invocations and has code size $C(A_i) = C_i(A)$, and another actor A_s for the subroutine call invocations, which has code size $C(A_s) = C_s(A)$.

Note that unlike the form of parenthesization handled by GDPPO for single appearance schedules, the parenthesization of arbitrary invocation sequences into optimally-compact loop hierarchies does not change the actor execution order, and thus, does not affect the buffering cost.

EXAMPLE 12. *Consider an SDF graph that consists of three actors* $\{X, Y, Z\}$, *and two edges* $\{(X, Y), (Y, Z)\}$ *such that* $\mathit{produced}((X, Y)) = \mathit{consumed}((Y, Z)) = 3$, *and* $\mathit{consumed}((X, Y)) = \mathit{produced}((Y, Z)) = 2$. *For clarity, assume that all invocations of each actor* V *are implemented with an inlined implementation having code size cost* $C(V)$. *Given the actor invocation sequence* (X, Y, X, Y, Z, Y, Z), *one of two parenthesizations leads to an optimum looping, depending on the relative code sizes of* X *and* Z. *If* $C(X) > C(Z)$, *then the optimum parenthesization is* $(XYXY)ZYZ$, *which leads to the looped schedule* $(2XY)ZYZ$; *if* $C(X) < C(Z)$, *however, the best parenthesization is* $XYX(YZYZ)$, *which leads to the looped schedule* $XYX(2YZ)$.

The CDPPO algorithm was first proposed in [3], where the algorithm was outlined as an interesting extension of GDPPO. In this paper, we introduce the first reported implementation of the CDPPO algorithm, and the application of the algorithm to a practical design space exploration problem. CDPPO computes an optimal parenthesization in a bottom-up fashion. Given, an SDF graph $G = (V, E)$ and an actor invocation sequence $f_1, f_2, \ldots, f_n$, where each $f_i \in V$, CDPPO first examines all 2-invocation *sub-chains* $(f_1, f_2), (f_2, f_3), \ldots, (f_{n-1}, f_n)$ to determine an optimally-compact looping structure (*subschedule*) for each of these sub-chains. For a 2-invocation sub-chain (f_i, f_{i+1}), the most compact subschedule is easily determined: if $f_i = f_{i+1}$, then $(2f_i)$ is the most compact subschedule, otherwise the original (unmodified) subschedule $f_i fi + 1$ is the most compact. After the optimal 2-node subschedules are computed in this manner, these subschedules are used to determine optimal 3-node subschedules (optimal looping structures for subschedules of the form f_i, f_{i+1}, f_{i+2}); the 2- and 3-node subschedules are then used to determine optimal 4-node subschedules, and so on until the n-node optimal subschedule is computed, which gives a minimum code size implementation of the input invocation sequence $f_1, f_2, \ldots, f_n$.

To understand the general approach for computing optimal subschedules given optimal subschedules for all "lower-order sub-chains," suppose that $\sigma = (f_i, f_{i+1}, \ldots, f_{i+k})$ is an arbitrary sub-chain of the input schedule $f_1, f_2, \ldots, f_n$ $(1 \leq k \leq (n - i))$, and suppose that optimal subschedules for all k-node sub-chains are available. An optimal subschedule for σ can be determined by first computing $C_j \equiv Z_{i,i+j} + Z_{(i+j+1),(i+k)}$, where $Z_{x,y}$ denotes the minimum code size cost for the sub-chain $(f_x, f_{x+1}, \ldots, f_y$. Any value of j that minimizes C_j gives an optimum point at which to "split" the sub-chain σ *if* $f_i, f_{i+1}, \ldots, f_{i+k}$ *are not to be executed through a single loop.*

Thus, it remains only to compute the minimum cost attainable for σ if the entire sub-chain is to be executed through a single loop. This minimum cost can be computed by determining those values of $m \in \mathit{divides}(k + 1)$, where $\mathit{divides}(k + 1)$ denotes the set of integers that divide the integer $(k + 1)$ (with zero remainder), for which σ is equivalent to the invocation sequence generated by a loop of the form $(mf_1 f_2 \ldots f(k + 1)/m)$. For each such m, the code size of the associated single-loop implementation can be expressed as $C_L + Z_{1,(k+1)/m}$, where C_L is the code size overhead of looping. If one or more values of m exist that give single-loop structures for implementing σ, then the associated code sizes are compared to determine an optimal single-loop structure, and this optimal single-loop structure is compared to the minimum value of C_j to determine an optimal looping structure for σ. Otherwise, the optimal looping structure for sigma is simply taken to be that corresponding to the minimum value of C_j.

The time-complexity of the overall CDPPO algorithm is $O(n^4)$, where n is the total number of actor invocations in the input schedule. Our initial experiments with CDPPO indicated that the run time of the algorithm can be high when the algorithm is applied to complex multirate systems. For example, when we applied our implementation of CDPPO to a minimum buffer schedule for a sample-rate conversion application, we observed a run time of 30 seconds. While such run times are acceptable for deterministic scheduling strategies in which CDPPO is applied only once, they are not acceptable in probabilistic algorithms, such as our evolutionary algorithm, that evaluate numerous candidate implementations, and invoke CDPPO for each candidate.

Thus, in our implementation of CDPPO, we have introduced two pairs of parameters α_1, α_2 and β_1, β_2 in the CDPPO algorithm to trade-off the accuracy of the optimization with the required run-time. These parameters are to be set experimentally — based on the specific SDF application under consideration — to yield the desired trade-off. The parameters α_1 and α_2 ($1 \leq \alpha_1 \leq \alpha_2$) specify the minimum and maximum sub-chain sizes for which optimal single-loop implementations (loop structures of the form $(mf_1f_2 \ldots f(k+1)/m)$) will be considered. Similarly, the algorithm parameters β_1 and β_2 specify the minimum and maximum sub-chain sizes for which alternative split positions will be evaluated. For sub-chain lengths less than β_1 or greater than β_2, the split will be taken to be halfway in the middle. We have found that the parameters $\alpha_1, \alpha_2, \beta_1, \beta_2$ provide an effective means for systematically trading-off between the run-time of CDPPO, and the code size cost of the resulting looping structures.

5. Basic Principles of 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. Mathematically, the concept of Pareto optimality can be defined as follows.

DEFINITION 3. *Let us consider, without loss of generality, a multi-objective minimization problem with m decision variables and n objec-*

tives:

$$\begin{array}{ll} \textit{Minimize} & \vec{y} = f(\vec{x}) = (f_1(\vec{x}), \ldots, f_n(\vec{x})) \\ \textit{subject to} & \vec{x} = (x_1, x_2, \ldots, x_m) \in X \\ & \vec{y} = (y_1, y_2, \ldots, y_n) \in Y \end{array} \tag{8}$$

where $\vec{x}$ *is called the* decision vector*,* X *is the* parameter space*,* $\vec{y}$ *is the* objective vector*, and* Y *is the* objective space*. A decision vector* $\vec{a} \in X$ *is said to* dominate *a decision vector* $\vec{b} \in X$ *(also written as* $\vec{a} \prec \vec{b}$*) iff*

$$\begin{array}{ll} \forall i \in \{1, \ldots, n\}: \ f_i(\vec{a}) \leq f_i(\vec{b}) & \wedge \\ \exists j \in \{1, \ldots, n\}: \ f_j(\vec{a}) < f_j(\vec{b}) & \end{array} \tag{9}$$

Additionally, in this study $\vec{a}$ *is said to* cover $\vec{b}$ *(*$\vec{a} \preceq \vec{b}$*) iff* $\vec{a} \prec \vec{b}$ *or* $f(\vec{a}) = f(\vec{b})$*. All decision vectors which are not dominated by any other decision vector of a given set are called* nondominated *regarding this set. The decision vectors that are nondominated within the entire search space are denoted as* Pareto optimal *and constitute the so-called* Pareto-optimal set*. The set of objective vectors that corresponds to the Pareto-optimal set forms the* Pareto-optimal front*.*

Evolutionary algorithms (EAs) seem to be especially suited to multi-objective optimization because they are able to capture several Pareto-optimal solutions in a single simulation run and may exploit similarities of solutions by recombination. Some researchers even suggest that multi-objective search and optimization might be a problem area where EAs do better than other blind search strategies [8]. The numerous applications and the rapidly growing interest in the area of multi-objective EAs supports this assumption. Moreover, up to now there are few if any alternatives to EA-based multi-objective optimization [10].

Several multi-objective EAs have been proposed since 1985, which mainly differ in fitness assignment scheme and the way of maintaining diversity in the population[5]. In this study, the Strength Pareto Evolutionary Algorithm (SPEA), a recent technique proposed in [17], is used. This algorithm has been shown to outperform other evolutionary approaches on different test problems [16, 17].

6. Evolutionary Algorithm

SPEA maintains besides the regular population an external set of individuals that represents the nondominated solutions of all solutions

[5] For reviews of multi-objective evolutionary algorithms, the interested reader is referred to [8, 10, 14]

generated so far. This set is updated every generation and afterwards included in the selection process. Therefore, nondominated solutions cannot get lost, which in turn also promotes diversity in the population.

The flow of the evolutionary algorithm used here is as follows.

Step 1: Generate an initial population P and create the empty external set P'.

Step 2: Copy nondominated members of P to P'.

Step 3: Remove solutions within P' which are covered by any other member of P'.

Step 4: Calculate the fitness of each individual in P as well as in P'.

Step 5: Select individuals from $P + P'$ (multiset union), until the mating pool P_{mate} is filled. In this study, binary tournament selection with replacement is used.

Step 6: Create the empty set P_{cross}. While the mating pool is not empty do

- Select two arbitrary individuals from P_{mate} without replacement.
- Create two children by recombining the two chosen parents.
- With probability p_c add the children to P_{cross}, otherwise add the parents to P_{cross}.

Step 7: Create the empty set P_{mut}. For each individual in P_{cross} do mutate it with mutation rate p_m and add it to P_{mut}.

Step 8: Set $P = P_{mut}$. If the maximum number of generations is reached then stop else go to Step 2.

Note that this presentation where selection, recombination, and mutation are considered as separate processes follows the one given in [1] and slightly differs from other formulations [9]. Nevertheless, both descriptions are mathematically equivalent.

6.1. Problem Coding

Each genotype consists of four parts which are encoded in separate chromosomes: i) schedule, ii) code model, iii) actor implementation vector, and iv) loop flag. This is illustrated in Figure 8.

Schedule	Code Model	Actor Implementation	Loop Flag
		A B	
A A B A B	2	0 1	1
	possible alleles: 0 = only subroutines 1 = only inlining 2 = mixed	possible alleles: 0 = subroutine 1 = inlining	possible alleles: 0 = looping off 1 = looping on

Figure 8. Example of an individual for the graph depicted in Figure 5a)

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 is based on a topological sort algorithm and simulates the execution of the actors:

Step 1: Create the empty topological sort T and initialize each arc α with $delay(\alpha)$ tokens.

Step 2: Choose the leftmost actor instance in the encoded actor sequence S which is fireable.

Step 3: Remove the selected actor instance at the corresponding position within S and append it to T.

Step 4: Simulate the firing of the chosen actor, i.e., from each ingoing arc α remove $consumed(\alpha)$ tokens and to each outgoing arc β add $produced(\beta)$ tokens.

Step 5: If S is not empty then go to Step 2 else T is the repaired schedule.

In contrast to a common topological sort procedure, the tie between several fireable actors is not broken at random but always the actor instance at the leftmost position within the encoded actor firing sequence is selected.

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. If the loop flag is set, the CDPPO algorithm (cf. Section 4) is applied to the flat schedule encoded in the individual, otherwise the schedule is not looped. As mentioned in Section 4, CDPPO has four parameters by which the run-time as well as the accuracy can be adjusted. These parameters are set by the user and fixed per optimization run.

6.2. Fitness Assignment

The fitness assignment procedure is a two-stage process. First, the individuals in the external set P' are ranked. Afterwards, the individuals in the population P are evaluated. In detail, the following three steps are performed:

Step 1: For each individual in the population and in the external set determine execution time, program memory, and data memory required by the encoded implementation.

Step 2: Each individual $i \in P'$ is assigned a real fitness value $f_i \in [0, 1)$ which is proportional to the number of population members $j \in P$ for which $i \preceq j$. Let n denote the number of individuals in P that are covered by i and assume N is the size of P. Then f_i is defined as

$$f_i = \frac{n}{N+1}.$$

Step 3: The fitness of an individual $j \in P$ is calculated by summing the fitness values of all externally stored individuals $i \in P'$ that cover j. We add one to the total in order to guarantee that members of P' have better fitness than members of P (note that fitness is to be minimized, i.e., small fitness values correspond to high reproduction probabilities):

$$f_j = 1 + \sum_{i, i \preceq j} f_i \quad \text{where } f_j \in [1, N).$$

6.3. Genetic Operators

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 [7], which do not destroy the permutation property; a short description of these operators is given below. Concerning the

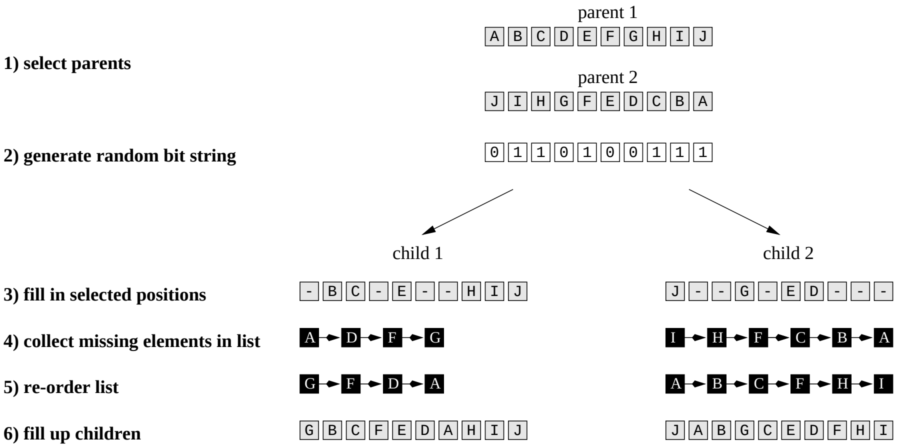

Figure 9. Order-based uniform crossover

other chromosomes, we work with one-point crossover and bit flip mutation [9] (as the code model gene is represented by an integer, mutation is done by choosing one of the three alleles at random).

6.3.1. *Order-based Uniform Crossover*

How this operator recombines two given permutations is illustrated in Figure 9. First, a bit string is randomly generated that is the same length as the parents. The bit positions correspond to the positions within the permutations defined by the parents. Then the first child is partially filled up. The node annotations of the first parent are copied to this child at the positions where the bit string contains a "1". Analogously, the second child inherits the node annotations of the second parent wherever a "0" occurs. Now, both children have gaps at complementary positions. In a third step, for each child a list is built up which contains all nodes not yet specified in the child. Afterwards, the list of the first child is sorted according to the node order in the second parent. Again, the same is done with the list of the second child. The relative order of the nodes in the list is identical to their relative order in the first parent. Finally, step by step and from left to right, the list elements are inserted at the gaps of the corresponding child.

EXAMPLE 13. *Let the first parent be the sequence* ABCDEFGHI *and the second parent the same sequence in reverse order. The intermediate results of the crossover phase are shown in Figure 9. At the end, two children,* GBCFEDAHIJ *and* JABGCEDFHI*, have been created.*

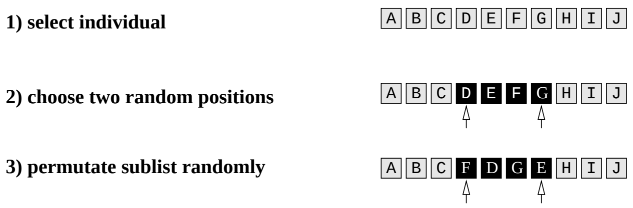

Figure 10. Scramble sublist mutation.

6.3.1.1. *Scramble Sublist Mutation* Mutation is done by permuting the elements between two selected positions, where both the positions and the subpermutation are chosen at random.

EXAMPLE 14. *Figure 10 shows an example for this operator. The chromosome* ABCDEFGHIJ *mutates to the sequence* ABCFDGEHIJ.

7. Experiments

The CDtoDAT example was used to compare the design spaces of the different DSP processors listed in Table I. For each of the real processors two kinds of experiments were performed: one time the parameters of CDPPO were set to $\alpha_1 = 1, \alpha_2 = \infty, \beta_1 = 10, \beta_2 = 40$ leading to suboptimal looping (about 5 hours run-time for one EA optimization run on a Sun ULTRA 30)[6], another time the focus was on optimal looping, where both accuracy and run-time of the CDPPO algorithm were maximum (the EA ran about 5 days).[7] For P1 only suboptimal looping was considered.

In all cases, the following EA parameters were used:

generations	:	250
population size	:	100
crossover rate	:	0.8
mutation rate	:	0.1[8]

Moreover, before every run a heuristic called APGAN (acyclic pairwise grouping of adjacent nodes [4]) was applied to this problem. The APGAN solution was inserted in two ways into the initial population:

[6] These CDPPO parameters were chosen based on preliminary experiments.

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

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

with and without looping. Finally, the set of all nondominated solutions found during the entire evolution process was considered as outcome of one single optimization run.

The nondominated fronts achieved by the EA in the different runs are shown in Figures 11 to 14. 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. Another extreme point (not shown in the figures) satisfies $D(S) = 1021$, but has only little overhead in the remaining two 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 Figures 11 and 13. 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 three real processors, one can observe that the ADSP 2106x produces less execution time overhead than the other processors which is in accordance with Table I. Subroutine calls are most frequently used in case of the TMS320C40 because of the high loop iteration overhead.

For processor P1 (Figure 14), 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.

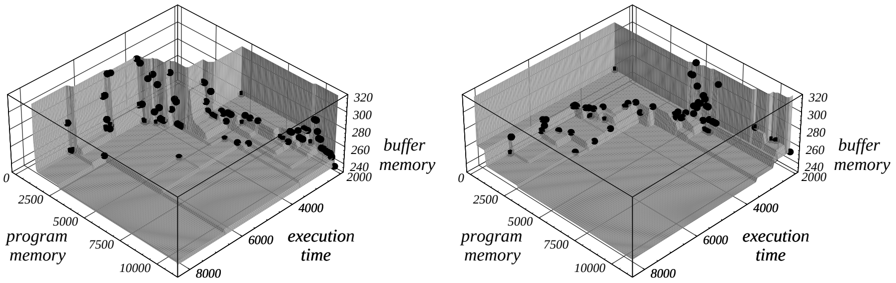

Figure 11. Motorola DSP56k (left: suboptimal looping, right: optimal looping)

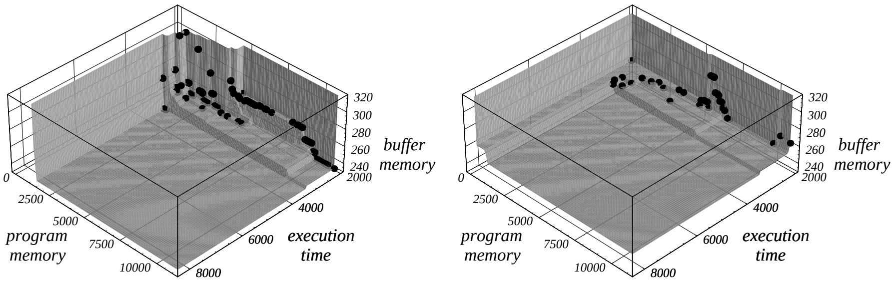

Figure 12. ADSP 2106x (left: suboptimal looping, right: optimal looping)

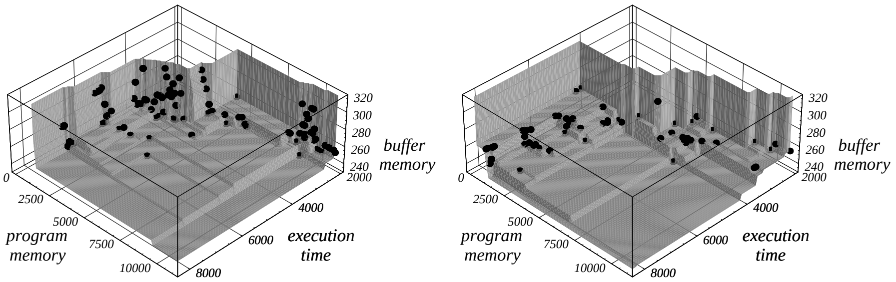

Figure 13. TI TMS320C40 (left: suboptimal looping, right: optimal looping)

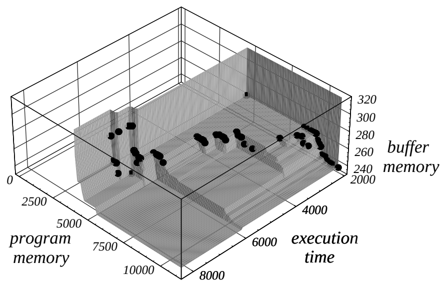

Figure 14. Processor P1 (suboptimal looping)

The effect of the looping algorithm on the obtained nondominated front can be clearly seen by comparing the results for suboptimal and optimal looping in Figures 11 to 13. In general, the nondominated solutions found require less buffer memory when the optimal looping algorithm is applied, the trade-off surface becomes much more flat in this dimension. It is also remarkable that for each real processor several solutions were generated that need less program memory than the implementation with lowest code size 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.

8. Conclusions

In this paper, we presented a methodology for exploring the design space of programmable digital signal processors (PDSP) implementations of synchronous dataflow graphs.

The design space is given here by schedule solutions that differ each in program memory requirements (code size), data memory requirements, and execution time. These solutions result from the freedom of allowing not only different execution orders, but also permitting looped schedules, and deciding whether for each actor appearance a subprogram call will be instantiated in the final code or whether the actor code is inlined.

With the above partially conflicting goals, optimal solutions are defined by Pareto optimality. The set of optimal solutions is explored using an evolutionary algorithm. For a number of well-known PDSP processors, the fronts of optimal solutions have been explored and compared. Also, we have shown that there is a trade-off between the quality of the found front of optimal solutions and the time spent in the used looping algorithm CDPPO (codesize dynamic programming post-optimization).

In the future, we'd like to investigate this trade-off between optimization time spent for exploration and within local search algorithms in more detail. Our code synthesis optimization problem seems to be a good candidate for studying these trade-offs.

References

1. Bäck, T.: 1996, *Evolutionary Algorithms in Theory and Practice*. New York, etc.: Oxford University Press.
2. Bhattacharyya, S., P. Murthy, and E. Lee: 1999, 'Synthesis of Embedded Software from Synchronous Dataflow Specifications'. *invited paper, J. of VLSI Signal Processing* p. to appear.
3. Bhattacharyya, S. S., P. K. Murthy, and E. A. Lee: 1995, 'Optimal Parenthesization of Lexical Orderings for DSP Block Diagrams'. In: *Proceedings of the International Workshop on VLSI Signal Processing*. Sakai, Osaka, Japan.
4. Bhattacharyya, S. S., P. K. Murthy, and E. A. Lee: 1996, *Software Synthesis from Dataflow Graphs*. Norwell, MA: Kluwer Academic Publishers.
5. Buck, J., S. Ha, E. Lee, and D. Messerschmitt: 1991, 'Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems'. *International Journal on Computer Simulation* **4**, 155–182.
6. Cormen, T. H., C. E. Leiserson, and R. L. Rivest: 1992, *Introduction to Algorithms*. MIT Press.
7. Davis, L.: 1991, *Handbook of Genetic Algorithms*, Chapt. 6, pp. 72–90. New York: Van Nostrand Reinhold.
8. Fonseca, C. M. and P. J. Fleming: 1995, 'An Overview of Evolutionary Algorithms in Multiobjective Optimization'. *Evolutionary Computation* **3**(1), 1–16.
9. Goldberg, D. E.: 1989, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, Massachusetts: Addison-Wesley.
10. Horn, J.: 1997, 'F1.9 Multicriteria decision making'. In: T. Bäck, D. B. Fogel, and Z. Michalewicz (eds.): *Handbook of Evolutionary Computation*. Bristol (UK): Institute of Physics Publishing.
11. Lauwereins, R., M. Engels, J. A. Peperstraete, E. Steegmans, and J. V. Ginderdeuren: 1990, 'GRAPE: A CASE Tool for Digital Signal Parallel Processing'. *IEEE ASSP Magazine* **7**(2), 32–43.
12. Lee, E. and D. Messerschmitt: 1987, 'Synchronous Dataflow'. *Proceedings of the IEEE* **75**(9), 1235–1245.
13. Ritz, S., M. Pankert, and H. Meyr: 1992, 'High level software synthesis for signal processing systems'. In: *Proc. Int. Conf. on Application-Specific Array Processors*. Berkeley, CA, pp. 679–693.
14. Tamaki, H., H. Kita, and S. Kobayashi: 1996, 'Multi-Objective Optimization by Genetic Algorithms: A Review'. In: *Proceedings of 1996 IEEE International*

Conference on Evolutionary Computation (ICEC'96). Piscataway, NJ, pp. 517–522.

15. Vaidyanathan, P. P.: 1993, *Multirate Systems and Filter Banks.* Prentice Hall.
16. Zitzler, E., K. Deb, and L. Thiele: 1999, 'Comparison of Multiobjective Evolutionary Algorithms: Empirical Results'. Technical Report 70, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich, Switzerland. Submitted to Evolutionary Computation Journal.
17. Zitzler, E. and L. Thiele: 1999, 'Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach'. *IEEE Transactions on Evolutionary Computation.* To appear.
18. Zivojnovic, V., J. Martinez, C. Schläger, and H. Meyr: 1994, 'A DSP-oriented Benchmarking Methodology'. In: *Int. Conf. on Sig. Proc: Applications & Technology.*