Optimized Software Synthesis for DSP Using Randomization Techniques (Revised Version of TIK Report 32)

Eckart Zitzler
Computer Engineering and Networks Laboratory (TIK)
Swiss Federal Institute of Technology (ETH)
Gloriastrasse 35, CH-8092 Zurich
Switzerland

Jürgen Teich
Computer Engineering
University of Paderborn
Warburger Str. 100, D-33098 Paderborn
Germany

Shuvra S. Bhattacharyya
Department of Electrical Engineering, and
Institute for Advanced Computer Studies (UMIACS)
University of Maryland
College Park MD 20742
U.S.A.

TIK Report No. 75
Institut für Technische Informatik und Kommunikationsnetze
ETH Zürich
Gloriastrasse 35
CH–8092 Zürich
Switzerland

July 1999

Contents

List of Figures

Abstract

This paper addresses the problem of trading-off between the minimization of program and data memory requirements of single-processor implementations of dataflow programs. Based on the formal model of synchronous data flow (SDF) graphs [20], so called single appearance schedules are known to be program-memory optimal. Among these schedules, buffer memory schedules are investigated and explored based on a two-step approach: (1) An Evolutionary Algorithm (EA) is applied to efficiently explore the (in general) exponential search space of actor firing orders. (2) For each order, the buffer costs are evaluated by applying a dynamic programming post-optimization step (GDPPO). This iterative approach is compared to existing heuristics for buffer memory optimization.

1 Introduction

In recent years, software synthesis has become an important component of the implementation process for embedded VLSI systems. This is due to important advantages of software such as flexibility, and shorter design time (faster time-to-market), which are critical considerations in many applications.

In dataflow, a specification consists of a directed graph in which the nodes represent computations and the edges specify the flow of data. A node is allowed to execute (fire) in case a certain firing rule is satisfied. If a node fires, it consumes a certain amount of data from the inputs and produces a certain amount of data on the outputs.

Synchronous dataflow [20] is a restricted form of dataflow in which the nodes, called *actors* have a simple firing rule: The number of data values (*tokens, samples*) produced and consumed by each actor is fixed and known at compile-time.

Example 1.1 *Figure 1 shows a simple SDF graph with four actors. A node is enabled for firing in case at least a fixed number of input tokens (as indicated by the edge annotation) has accumulated on each input edge. When the actor executes, a fixed number of output tokens are produced at its outgoing edges which is also a constant specified at compile-time and annotated at the tail of each outgoing edge.*

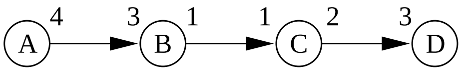


Figure 1: A simple SDF graph.

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, the SDF model is used in industrial DSP design tools, e.g., SPW by Cadence, COSSAP (now) by Synopsys, and Advanced Design System from Hewlett-Packard, as well as in research-oriented environments, e.g., Ptolemy [8], GRAPE [19], and COSSAP [25]. Those systems include code generation tools with code (usually optimized assembly code) stored for each actor in a target-specific library. Typically, code is generated from a given schedule by instantiating actor code in the final program. Subroutine calls may have unacceptable overhead, especially if there are many small tasks. Hence, a code generation method that generates inline code is often assumed.

With this model, it is evident that the size of the required program memory strongly depends on the number of times an actor appears in a schedule. So called *single appearance schedules*, where each actor appears only once (however possibly embedded in a nested loop) in a schedule, are evidently program memory optimal under this model of inline code generation. Results on the existence of such schedules have already been published for general SDF graphs [3]. For acyclic graphs, there always exists at least one single appearance schedule.

In this paper, we treat the problem of generating single appearance schedules that minimize the amount of required buffer memory for the class of acyclic SDF graphs. Such a methodology may be considered as part of a general framework that considers general SDF graphs and generates schedules for acyclic subgraphs using our approach. In particular, necessary and sufficient conditions for the existence of single appearance schedules for general SDF graphs and efficient algorithms for computing them have been given in [4, 5]. These techniques require decomposing each strongly connected component into an acyclic graph that consists of *clusters* of smaller strongly connected components, constructing a single appearance schedule for this acyclic graph, and then recursively applying this procedure to each clustered strongly connected component to obtain the subschedule for the corresponding cluster.

1.1 Motivation

Given is an acyclic SDF graph in the following. The number of single appearance schedules that must be investigated is at least equal to (and often much greater than) the number of topological sorts of actors in the graph. Note that this number may be exponential in the size of the graph; e.g., a complete bipartite graph with $2n$ nodes has $(n!)^2$ possible topological sorts. This complexity prevents techniques based on enumeration from being applicable.

In [7], a heuristic called APGAN is introduced that is used inside state of the art design systems such as Ptolemy or SPW. It constructs a schedule with the objective to minimize buffer memory. This procedure of low polynomial time complexity has been shown to give optimal results for a certain class of graphs having relatively regular structure. Also, a complementary procedure called RPMC (also part of Ptolemy) that works well on more irregular graph structures is discussed and compared to our approach here.

Experiments show that, although being computationally efficient, these heuristics can sometimes produce results that are far from optimal. Even simple testcases may be constructed where the performance (buffer cost) obtained by applying these heuristics differ from the global minimum by more than 2000%, see graph no. 2 in Example 1.2.

Example 1.2 *We consider two testgraphs and compare different buffer optimization algorithms (see Table 1.2). The first graph with 10 nodes is shown also in Fig. 2. For this rather small and simple graph, already 362 880 different topological sorts (actor firing orders) may be constructed. The minimal buffer memory requirement has been evaluated to 3003 whereas for the worst topological sort, the minimal costs were computed as 15 705 memory units.*

The second graph is a randomly generated graph with 50 nodes. In the table, 4 different methods are compared with respect to the best cost found and the amount of required CPU-time. The first method uses an Evolutionary Algorithm (EA) that

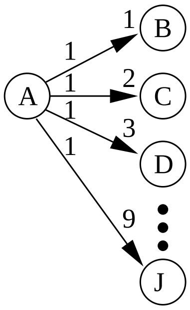


Figure 2: A simple SDF graph.

performs 3000 fitness calculations, the second is the APGAN heuristic, the third is the RPMC heuristic, the fourth is a Monte Carlo *simulation (3000 random tries) and the fifth an exhaustive search procedure which did not terminate in the second case.*

Graph #	method	best cost (units)	runtime (s)
1	EA	3003	4.57
1	APGAN	3015	0.02
1	RPMC	3151	0.03
1	Monte Carlo	3014	3.3
1	Exhaust. Search	3003	373
2	EA	669 380	527.87
2	APGAN	15 063 956	1.88
2	RPMC	1 378 112	2.03
2	Monte Carlo	2 600 349	340.66
2	Exhaust. Search	?	?

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

The motivation of the following work was to develop a methodology that has the following features:

- *Cost-competitiveness:* the optimization procedure should provide solutions which provide the same or lower buffering costs as the heuristics APGAN and RPMC in most investigated test cases.
- *Run-time tolerability:* in embedded DSP applications, compilers are allowed to spend more time for optimization of code as in general-purpose compilers, because code-optimality is critical [23]. Hence, compilation times in the range from 1s to 100s, sometimes even in the order of minutes, still seem to be tolerable.

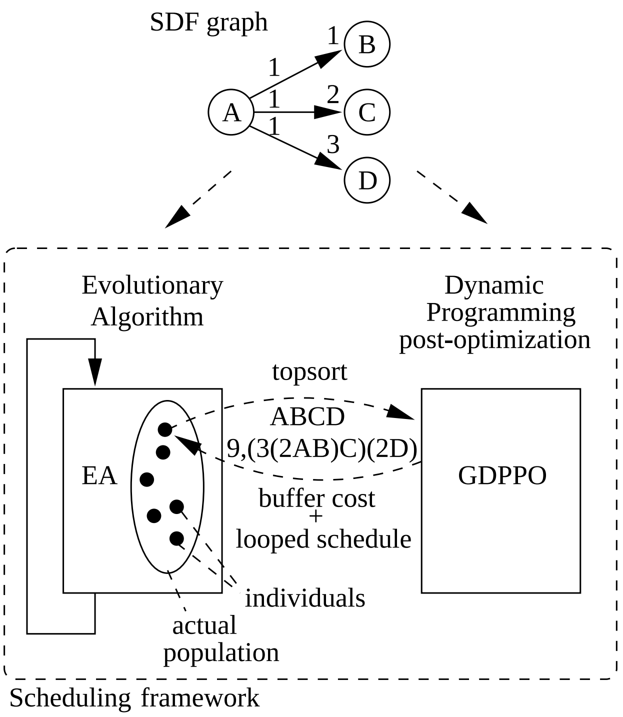


Figure 3: Overview of the scheduling framework using Evolutionary Algorithms and Dynamic Programming (GDPPO: generalized dynamic programming post optimization for optimally parenthesizing actor orderings [24], discussed further in Section 3.2) for constructing buffer memory optimal schedules.

Therefore, it does not seem reasonable to compare different algorithms in terms of the quality (buffer memory cost) obtained during the run-time limit of the fastest heuristic. Instead, we compare the quality obtained during a *tolerable run-time limit*, say, e.g., one minute of CPU-time.

1.2 Proposed Approach

Here, we use a unique two-step approach to find buffer-minimal schedules:

- An Evolutionary Algorithm (EA) is used to efficiently explore the space of topological sorts of actors given an SDF graph using a population of N individuals where each individual of a *population* encodes a topological sort.
- For each topological sort, a buffer optimal schedule is constructed based on a well-known dynamic programming post optimization step [24] that determines a loop nest by parenthesization (see Fig. 3) that is buffer cost optimal (for the given topological order of actors). The run-time of this optimization step is $\mathcal{O}(N^3)$.

Example 1.3 *The overall picture of the scheduling framework is depicted in Fig. 3. The approach is called GASAS (for Genetic Algorithm exploration of Single Ap-*

pearance Schedules). The EA iteratively transforms a population of individuals, each of which is coding a topological sort of the actors of a given SDF graph. The buffer cost of each individual is evaluated by calling a Dynamic Programming post-optimizer that performs a parenthesization of the proposed actor firing order to obtain a buffer-optimal schedule (for the given order of actors). The evaluated optimal costs are returned to the EA that applies transformations to the individuals in order to find better topological sorts. Details on the optimization procedure and the cost function will be explained in the following section. The total run-time of the algorithm is $\mathcal{O}(Z\ N^3)$ *where* Z *is the number of evocations of the dynamic program post-optimizer.*

1.3 Related Work

The interaction between instruction scheduling and register allocation in procedural language compilers has been studied extensively [16, 1], and optimal management of this interaction has been shown to be intractable [14]. More recently, the issue of optimal storage allocation has been examined in the context of high-level synthesis for iterative DSP programs [11], and code generation for embedded processors that have highly irregular instruction formats and register sets [23, 17]. For irregular embedded processors, code generation techniques have also been developed for the purpose of minimizing the amount of memory required to store a program's code [21, 22]. However, because of their focus on fine-grain scheduling, the above efforts apply to a homogeneous data flow model – that is, a model in which each computation (dataflow vertex) produces and consumes a single value to/from each incident edge. In particular these efforts do not address the challenges of keeping code size costs manageable in general SDF graphs, in which actor production and consumption parameters may be arbitrary.

In practice, fine-grain scheduling techniques, such as those described above, are complementary to the SDF techniques that we present in this paper. This is because SDF specifications are typically at a coarse-grain level. Individual actors can be of arbitrary complexity. Typically they range in complexity from basic operations such as addition or multiplication to signal processing subsystems such as FFT units or adaptive filters. The techniques developed in this paper primarily affect sequencing and communication at the inter-actor level. In SDF-based design environments, the internal functionality of individual actors is typically specified in assembly language or an imperative, high-level language such as C. Thus fine-grained scheduling algorithms, such as those mentioned above can be used to optimize the mapping of intra-actor functionality to the target processor, while the SDF scheduling techniques of this paper are used to optimize the target code at the inter-actor level.

Similarly, Fabri [12] and others have examined the problem of managing pools of logical buffers that have varying sizes, given a set of buffer lifetimes, but such efforts are also in isolation of the scheduling problems that we face in the context of general SDF graphs.

Preliminary ideas of our work can be found in [27] and [28].

1.4 Overview

We conclude this section with a summary of what follows.

In Section 2, the required notation is introduced, and the overall methodology is explained in more detail in Section 3. Also, the cost model for buffer cost and the role of the dynamic program post-optimizer are outlined.

Section 4 includes an explanation of why an Evolutionary Algorithm is favorable for design space exploration and describes the Evolutionary Algorithm approach. In particular, we describe how topological sorts for SDF graphs may be efficiently generated and explored. Details on the chosen genetic operators and the coding mechanism are provided.

Finally, a quantitative comparison of this new approach with existing algorithms like APGAN and RPMC [7] as well as with other probabilistic optimization methods like *Monte Carlo* and *Hill Climbing*, and a randomized version of APGAN called RAPGAN (cf. Section 5) are described in Section 6. In case a run-time limit of 100 seconds for the exploration of the search space is allowed, the Evolutionary Algorithm approach beats existing heuristics in almost 100% of the test cases which have been chosen from a library of existing problems and a set of randomly generated test graphs.

2 The SDF-scheduling framework

2.1 Background and Notation

Synchronous dataflow [20] is a restricted form of data flow in which nodes of a directed graph represent actors, consume a fixed amount of data items (tokens, samples) per invocation and produce a fixed amount of output samples per invocation.

Definition 2.1 (SDF graph) *An SDF graph G denotes a 5-tuple $G = (V, A, produced, consumed, delay)$ where*

- *V is the set of nodes (*actors*) ($V = \{v_1, v_2, \cdots, v_{\mathrm{K}}\}$).*
- *A is the set of directed edges. With $source(\alpha)$ ($sink(\alpha)$), we denote the source node (target node) of an edge $\alpha \in A$.*
- *$produced : A \rightarrow \mathbf{N}$ denotes a function that assigns to each directed edge $\alpha \in A$ the number of produced tokens $produced(\alpha)$ per invocation of actor $source(\alpha)$.*
- *$consumed : A \rightarrow \mathbf{N}$ denotes a function that assigns to each directed edge $\alpha \in A$ the number of consumed tokens per invocation of actor $sink(\alpha)$.*
- *$delay : A \rightarrow \mathbf{N}_0$ denotes the function that assigns to each edge $\alpha \in A$ the number of initial tokens $delay(\alpha)$.*

A *schedule* is a sequence of actor firings. A properly-constructed SDF graph is compiled by first constructing a finite schedule S that fires each actor at least once, does not deadlock, and produces no net change in the number of tokens queues associated with each edge. 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 [20]. We represent these minimum numbers of firings by a function q_{G} or simply q in case G is known from the context with $q : V \to \mathbf{N}$.

Example 2.1 *Figure 1 shows an example of an SDF graph with four nodes and two edges. The four nodes (actors) are labeled A, B, C, D, respectively. The graph 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 $q(A) = 9$, $q(B) = q(C) = 12$, $q(D) = 8$. The schedule*

$$(\infty(2ABC)DABCDBC(2ABCD)A(2BC)(2ABC)A(2BCD))$$

represents a valid schedule for the SDF graph shown in Fig. 1. Here, a parenthesized term $(n\ S_1\ S_2\ \cdots\ S_k)$ specifies n successive firings of the "subschedule" $S_1\ S_2\ \cdots\ S_k$.

Each parenthesized term $(n\ S_1\ S_2\ \cdots\ S_k)$ is referred to as *schedule loop* having *iteration count* n and *iterands* $S_1, S_2, \cdots, S_k$. We say that a schedule for an SDF graph is a *looped schedule* if it contains zero or more schedule loops. Thus, the *looped* qualification indicates that the schedule in question may contain one or more schedule loops, but the existence of a loop is not required.

A schedule is called *single appearance schedule* if it contains only one appearance of each actor.

Example 2.2 *The schedule $(\infty(9A)(12B)(12C)(8D))$ is a valid single appearance schedule for the graph shown in Fig. 1.*

In general, a schedule of the form

$$(\infty\ (q(N_1)N_1)\ (q(N_2)N_2)\ \cdots\ (q(N_{\mathrm{K}})N_{\mathrm{K}}))$$

where N_i denotes the (label of the) ith node of a given SDF graph, and K denotes the number of nodes of the given graph, is called *flat single appearance schedule.*

2.2 Code generation model

We consider the problem of code generation by code inlining given a SDF graph specification while considering single processor implementations: Corresponding to

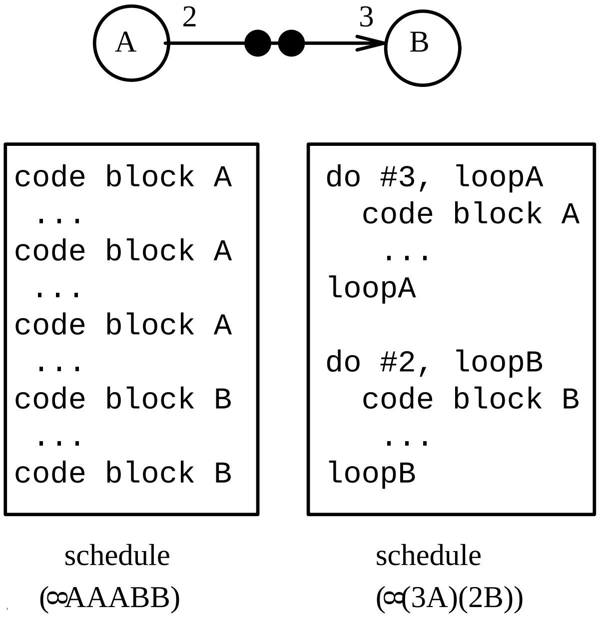


Figure 4: Code generation by inlining.

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

Example 2.3 *Given is the simple SDF graph in Fig. 4. A valid schedule for this graph is* $(\infty AAABB)$*. Also valid is the single appearance looped schedule* $(\infty(3A)(2B))$*. For both schedules, it is shown how the code generation by inlining works to generate the final code.*

Implied by this model of code generation, any valid single appearance schedule gives the minimum code space (program memory) cost. This approximation, however, neglects loop overhead. In practical SDF models of applications, actors are usually DSP subsystems of medium to large granularity, and the code size overhead of a loop is typically small compared to the size of individual code blocks. Thus, neglecting loop overhead does not lead to misleading results in our context.

2.3 Buffer cost model

The simplest model for buffering (data memory) is to assume that a distinct area of memory is allocated for each edge of a given graph. In order to determine the amount of data needed to store the tokens that accumulate on each edge during the evolution of a schedule S, we define the cost function

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

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

Example 2.4 *Consider the schedule* $(\infty(9A)(12B)(12C)(8D))$ *for the SDF graph shown in Fig. 1. This schedule has a buffer memory requirement of* $36 + 12 + 24 = 72$. *Similarly, the buffer memory requirement of the schedule* $(\infty(3(3A)(4B))(4(3C)(2D)))$ *is* $12 + 12 + 6 = 30$.

Another model for buffering is to use a shared buffer of size

$$\max_{i=1}^{K-1}(q(N_i)\ produced(N_i)), \tag{2}$$

which gives the maximum amount of data transferred on any edge in one period (one iteration of the outermost loop) of the flat single appearance schedule. In case of nested loops, however, the use of shared buffers may be awkward. Also, the management of pointers and the handling of initial delays on edges require special attention. In the latter case, there is often no logical place in the buffer to place the delays since the entire buffer might be written over by the time we reach the actor that consumes the delays. Therefore, we will use the non-shared buffer model for all edges with non-zero initial delays, and decide on the usefulness of buffer sharing for edges without delays.

2.4 Buffer Memory Optimization

The following lower bounds on the buffer memory requirements have been published in [7]:

Definition 2.2 (Buffer memory lower bound) *A* buffer memory lower bound *(BMLB) of an SDF edge* $\alpha \in A$, *denoted* $BMLB(\alpha)$, *is given by*

$$BMLB(\alpha) = \begin{cases} (\eta(\alpha) + delay(\alpha)) & \text{if } (delay(\alpha) < \eta(\alpha)) \\ delay(\alpha) & \text{if } (delay(\alpha) \geq \eta(\alpha)) \end{cases} \tag{3}$$

where

$$\eta(\alpha) = \frac{produced(\alpha)\ consumed(\alpha)}{\gcd(\{produced(\alpha), consumed(\alpha)\})}.$$

If G *is an SDF graph, then*

$$\left(\sum_{\alpha \in A} BMLB(\alpha)\right)$$

is called the BMLB of G, *and a valid single appearance schedule* S *for* G *that satisfies* $max\_tokens(\alpha, S) = BMLB(\alpha)\ \forall \alpha \in A$ *is called* BMLB schedule *for* G.

It should be noted that not all SDF graphs have valid BMLB schedules [6], but many practical graphs do.

3 An Evolutionary Approach for Memory Optimization

In this section, we describe the sharing of work for exploring the search space of buffer memory optimal single appearance schedule solutions.

The separation of work has already been figured out in Fig. 3.

3.1 Exploration of topological sorts using the EA

Here, given an acyclic SDF graph, the main difficulties consist in finding a coding of feasible topological sorts. One could use a coding scheme which represents a permutation of the actor set. However, by genetic mutation and crossover, the permutations in general would not represent topological sorts. Penalty functions that punish infeasible permutations would not prevent the needle-in-the-haystack search in some cases. Hence, a simple repair mechanism must be used in order to guarantee each individual in the actual population to represent a topological sort.

Also, the simple extension to also allow permutations where each actor may fire enough times without being in topological order are possible. This happens in cases where enough delays are accumulated on corresponding input edges of an actor in question.

Details on the coding scheme are given in the Section 4.2 that deals with all implementation issues of the evolutionary search procedure. Instead, we conclude Section 3 by summarizing the way, a provably buffer-memory optimal schedule may be found for a given actor ordering by dynamic programming.

3.2 Dynamic Programming Post Optimization

Given a consistent acyclic SDF graph G and, for simplicity, assume that each edge $\alpha \in A$ satisfies $delay(\alpha) = 0$ (delayless edges).

In [24], it has been shown that given a topological sort of actors of a consistent, delayless and acyclic SDF graph, a minimum buffer memory schedule over all single-appearance schedules for this graph with the same lexical ordering as the topological sort may be determined as the solution of a dynamic programming problem (DPPO).[1]

Example 3.1 *Consider again the SDF graph in Fig. 1. With $q(A) = 9$, $q(B) = q(C) = 12$, and $q(D) = 8$, an optimal schedule is $(\infty(3(3A)(4B))(4(3C)\ (2D)))$ with a buffer cost of 30. Given the topological order of nodes A, B, C, D as imposed by the edges of G, this schedule is obtained by parenthesization of the string. Note that this optimal schedule contains a break in the chain at some actor k, $1 \leq k \leq \mathrm{K} - 1$.*

[1]The extension GDPPO in [6] guarantees that given any (not necessarily delayless) consistent SDF graph and a lexical ordering (not necessarily a topological sort), a single appearance schedule is computed that minimizes the buffer memory over all single appearance schedules that have the given lexical ordering (assuming that at least one valid single appearance schedule exists that has the given lexical ordering).

Because the parenthesization is optimal, the chains to the left of k and to the right of k must also be parenthesized optimally. This structure of the optimization problem is essential for dynamic programming.

Let $b[i,j]$, $1 \leq i \leq j \leq \mathrm{K}$ denote the minimum buffer cost (over all valid single appearance schedules with the same actor ordering) for scheduling the subgraph induced by the actors $v_i, v_{i+1}, \cdots, v_j$. [2]

For $1 \leq i \leq j < \mathrm{K}$, $b[i,j]$ may be calculated as

$$b[i,j] = \min_{i \leq k < j} \{b[i,k] + b[k+1,j] + c_{i,j}[k]\}, \tag{4}$$

where $b[i,k]$ is the minimum buffer cost for the subgraph induced by $\{v_i, \cdots, v_k\}$ and $b[k+1,j]$ is the minimum buffer cost for the subgraph induced by $\{v_{k+1}, \cdots, v_j\}$, $b[i,i] = 0$, and $c_{i,j}[k]$ denotes the memory cost at the split if we split the graph at actor v_k into a parenthesized left subgraph induced by $\{v_i, \cdots, v_k\}$ and a parenthesized right subgraph induced by $\{v_{k+1}, \cdots, v_j\}$. The minimal buffer cost of the complete graph is then obtained as $b[1, \mathrm{K}]$.[3]

The split cost $c_{i,j}[k]$ is obtained as

$$c_{i,j}[k] = \frac{\sum_{\alpha \in A \cap A_{cut}} (produced(\alpha)\ q(source(\alpha))}{\gcd(\{q(v_i), q(v_{i+1}), \cdots, q(v_j)\})} \tag{5}$$

where A_{cut} denotes the edge set

$$A_{cut} = \{\beta \in A | (source(\beta) \in \{v_i, v_{i+1}, \cdots, v_k\} \text{ and } sink(\beta) \in \{v_{k+1}, v_{k+2}, \cdots, v_j\})\} \tag{6}$$

An easy extension to also include shared buffering is to assume that there is one shared buffer of size

$$cs_{i,j} = \frac{\max_{i \leq k < j} (\sum_{\alpha \in A \cap A_{cut}} (produced(\alpha)\ q(source(\alpha)))}{\gcd(\{q(v_i), q(v_{i+1}), \cdots, q(v_j)\})} \tag{7}$$

for the subgraph with nodes $\{v_i, v_{i+1}, \cdots, v_j\}$. Finally, the cost $b[i,j]$ is replaced by $b'[i,j]$ with

$$b'[i,j] = \min(\{b[i,j], cs_{i,j}\})$$

In this case, the optimization procedure also determines which edges to be implemented in a shared buffer and which edges should be implemented separately.

An extension to also include initial delay on edges is also possible.

[2]This graph has the node set $V' = \{v_i, v_{i+1}, \cdots, v_j\}$ and the edge set $A' = \{\alpha \in A | source(\alpha) \in V' \text{ and } sink(\alpha) \in V'\}$.

[3]In the case of general acyclic graphs with multiple source and/or multiple sink nodes, a dummy node v_0 that becomes the unique source of all former source nodes may be introduced. For multiple sinks, a dummy node $n_{\mathrm{K}+1}$ may be similarly introduced that becomes target of all former sink nodes. Finally compute the minimum buffer cost as $b[0, \mathrm{K}+1]$.

4 The Evolutionary Algorithm

Evolutionary Algorithm (EA) is used as an umbrella term for a class of computational models that mimic natural evolution in order to solve arbitrary optimization problems. Since the 1970's, several evolutionary methodologies have been proposed, mainly genetic algorithms, evolutionary programming, and evolution strategies [2]. All these approaches have in common that they process a set of candidate solutions, called *population*, simultaneously. Using strong simplifications, this set is subsequently modified by the two basic principles of evolution: *selection* and *variation*. Selection represents the competition for resources among living beings. Some are better than others and more likely to survive and to reproduce their genetic information. In Evolutionary Algorithms, natural selection is simulated by a stochastic selection process. Each solution (*individual*) is given a chance to reproduce a certain number of times, proportionate to their quality. Thereby, quality is assessed by evaluating the individuals and assigning them scalar *fitness* values. The other principle, variation, imitates natural capability of creating "new" living beings by means of crossing over and mutation.

Although simplistic from a biologist's viewpoint, these algorithms have proven themselves as a general, robust, and powerful search mechanism.

4.1 Motivation for Using an Evolutionary Algorithm

From Example 1.2, it becomes clear that there exist simple graphs for which there is a big gap between the quality of solution obtained using different heuristics such as APGAN and an Evolutionary Algorithm (EA).

Also, the example has shown that an Evolutionary Algorithm for efficiently exploring the vast search space of topological sorts has provided much better solutions than just performing a random search (Monte-Carlo simulation). If the run-time of such an iterative approach is still affordable, a performance gap of several orders of magnitude may be avoided.

Moreover, an evolutionary algorithm is here a promising choice for the following reasons:

- Topological sorts may be easily coded using an Evolutionary Algorithm. Details on the coding scheme are given in Section 4.2.
- Evolutionary algorithms perform a parallel sampling of the search space by working on populations of individuals.
- The optimization function is allowed to be non-linear and arbitrarily complex.
- Like other probabilistic search approaches, evolutionary algorithms can meaningfully exploit whatever level of compile-time tolerability is available to an embedded system designer. In contrast, deterministic algorithms take a fixed amount of time on a given platform, and cannot make use of additional "computational energy" that is available.

- Evolutionary algorithms provide a flexible, robust search strategy that has proven to be applicable for a large number of useful problems.

4.2 Structure of the Evolutionary Algorithm

As described previously, we use an Evolutionary Algorithm to sample the search space of all possible topological sorts of a given SDF graph. Each topological sort is assigned a fitness value, reflecting buffer memory requirements, which is computed by the SDF framework. The aim is to find the graph node order which yields a single appearance schedule with minimal buffer cost among all possible orders.

The flowchart of the Evolutionary Algorithm is depicted in Figure 5. The left hand side shows the single steps of the algorithm, the right hand side visualizes the effect of each step on the population. The contents of the population are exemplary and refer to the simple SDF graph presented in Figure 1.

In the first step an initial population is created. It contains randomly generated individuals, the *phenotype* of which represents a topological sort. The evolution of new generations is done in the main loop. After the fitness evaluation, the population is modified by applying the three genetic operators *selection*, *recombination*, and *mutation*. While the selection operator increases the average quality of the population, the two variation methods recombination and mutation serve the exploration of new topological sorts. In contrast to other descriptions of Evolutionary Algorithms [15][18], here we view selection, recombination, and mutation as independent processes subsequently working on the whole population.

In the following, we deal with the details of the Evolutionary Algorithm. The problem of encoding node orders and transforming arbitrary orders into valid topological sorts is addressed in Section 4.3. Section 4.4 treats the kind of genetic operators we have implemented, while Section 4.5 examines the influence of different crossover and mutation probabilities on the performance of the Evolutionary Algorithm.

4.3 Coding and Repair Mechanism

Our problem is a combinatorial optimization problem rather than a numerical optimization problem. It naturally suggests to use an order-based representation. But we do not consider arbitrary orderings of the nodes in the graph because we are only interested in topological sorts. Instead, each individual encodes a permutation over the set of nodes. Hence, the initial population consists of randomly created permutations.

Beyond it, a simple repair mechanism, transforming permutations into topological sorts, guarantees that every genotype can be mapped to a valid topological sort. So, there are no infeasible individuals in the population. On the other hand, since each topological sort is simultaneously a permutation, the whole search space is covered by this representation.

The skeleton of the repair algorithm is formed by a common algorithm for sorting graphs topologically (see Figure 6). In each step, a node with an indegree equal

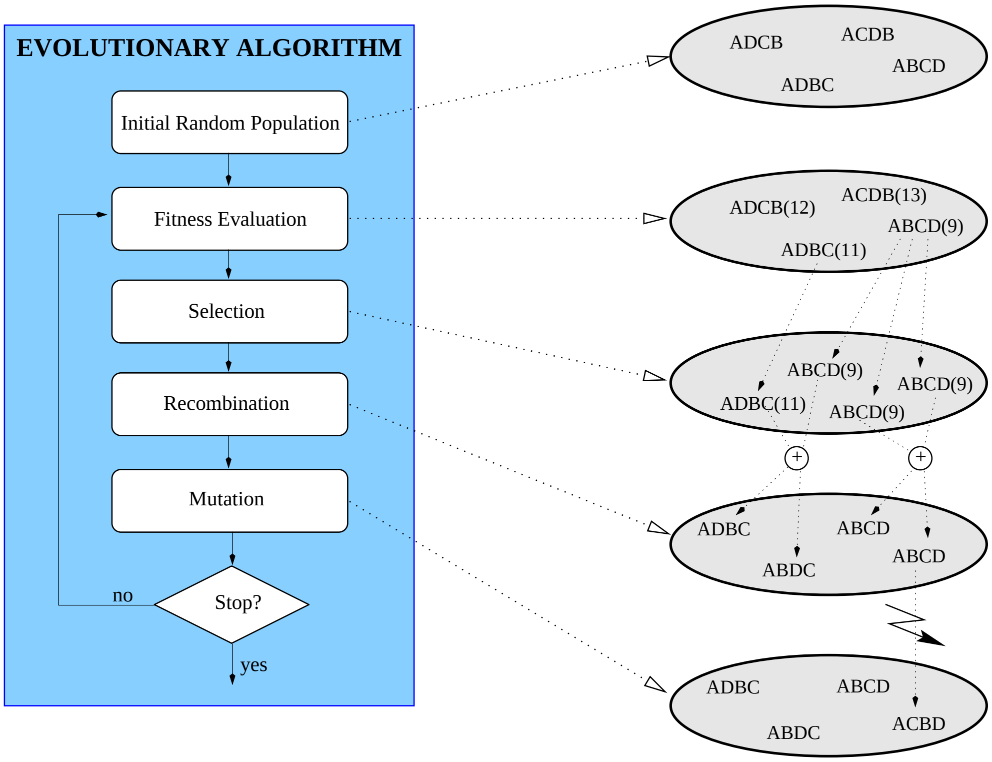


Figure 5: Flowchart of the Evolutionary Algorithm.

to zero is chosen and removed from the graph (together with the incident edges). The order in which the nodes appear determines the topological sort. With it, the tie between several nodes with no incoming edges is normally broken at random. Our algorithm, however, always selects the node at the leftmost position within the permutation. This ensures on the one hand, that each individual is mapped unambiguously to one topological sort, and, on the other hand, that every topological sort has at least one encoding.

Example 4.1 *Recall the SDF graph depicted in Figure 2, and suppose, the repair algorithm is working on the permutation* BCDEFAGHIJ*. Since the node* A *has no incoming edges but is predecessor of all other nodes, it has to be placed first in any topological sort. The order of the remaining nodes is unchanged. Therefore, the resulting topological sort after the repair procedure in Figure 6 is* ABCDEFGHIJ*.*

Note that in the real system an optimized algorithm has been implemented which avoids delete operations on the SDF graph.

4.4 Genetic Operators

Many selection schemes exist as well as crossover and mutation methods specialized for order-based representations. In the following, we outline the operators we have chosen.

```
PROCEDURE Repair
        IN:   permutation : node_list;
              graph : sdf_graph;
        OUT:  topsort : node_list;
BEGIN
        Clear(topsort);
        WHILE NOT Empty(permutation) DO
              node := First(permutation);
              WHILE node <> NIL AND Indegree(node, graph) > 0 DO
                    node := Next(node, permutation);
              OD
              IF node = NIL THEN
                    Error("cyclic graph");
                    Stop;
              FI
              Remove(node, permutation);
              Append(node, topsort);
              FOR edge IN OutgoingEdges(node, graph) DO
                    DeleteEdge(edge, graph);
              OD
        OD
END
```

Figure 6: Repair algorithm for mapping permutations to topological sorts.

Selection The selection scheme chosen is *tournament selection*. In tournament selection a fixed number of individuals is picked out randomly, and the individual having the best fitness value (lowest buffer cost) within this group is copied to the new population. This process is repeated until the new population has been filled up.

The advantage of this algorithm is its linear time complexity $\mathcal{O}(N)$, where N denotes the population size. Fitness proportionate selection has linear time complexity too, but in contrast to tournament selection it is not translation invariant [10]. That means increasing the fitness values by adding a constant causes a change in the result of the selection process.[4] On the other hand, e.g., rank selection, which is translation invariant, needs al least $\mathcal{O}(N \log N)$ run-time due to sorting the population.

The issue of run-time is important for our concerns. Although in Section 1.1, run-time tolerability was claimed as an essential feature of our methodology, we paid special attention to a fast implementation of the Evolutionary Algorithm in order

[4]Since the buffer costs may vary extremely for different applications, translation invariance is an important aspect.

to be competitive to the existing, deterministic heuristics.

Independent of this selection scheme, an *elitist strategy* has been implemented: the best individual per generation is preserved by simply copying it to the population of the next generation.

Crossover Since individuals encode permutations, it is crucial that the variation operators do not destroy the permutation property. Each node has to appear exactly one time in a sequence coded by a chromosome. To accomplish this task, the uniform order-based crossover operator [9][13] is applied. We give a short description of it below.

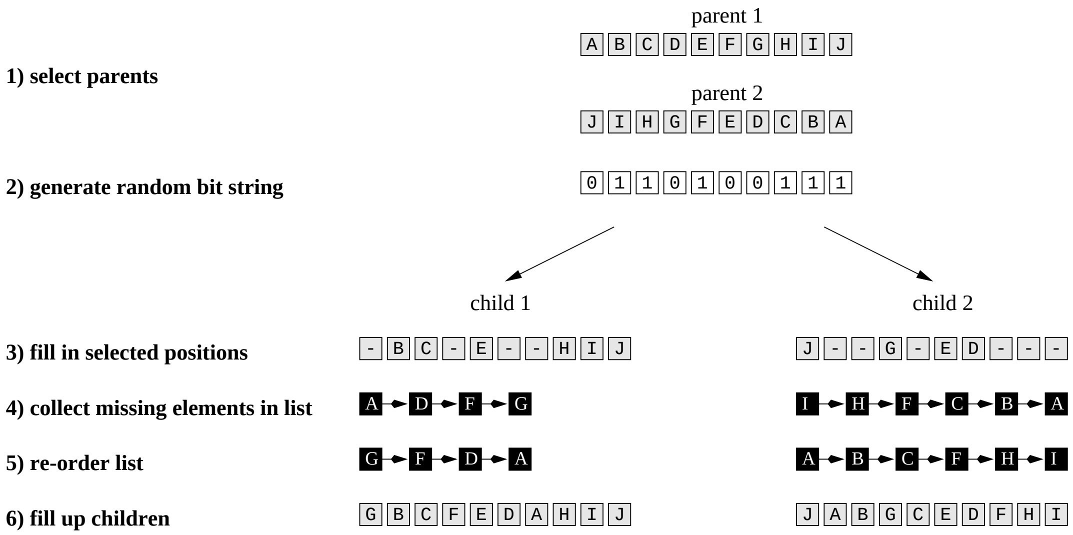


Figure 7: Uniform order-based crossover.

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 partly 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 to 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 4.2 *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 7. At the end, two children,* GBCFEDAHIJ *and* JABGCEDFHI*, have been created.*

Mutation Mutation is done by permuting the elements between two selected positions, whereas both the positions and the subpermutation are chosen by random. That is what Davis calls *scramble sublist mutation* [9].

Example 4.3 *Figure 8 shows an example for this operator. The chromosome* ABCDEFGHIJ *mutates to the sequence* ABCFDGEHIJ.

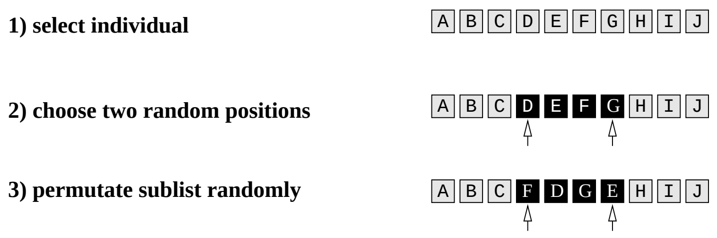


Figure 8: Scramble sublist mutation.

4.5 Crossover Probability and Mutation Probability

To the recombination operator as well as to the mutation operator, probabilities for their application are associated, namely the crossover probability p_c and the mutation probability p_m. The setting of these two parameters might have great influence on the outcome and the convergence speed of the Evolutionary Algorithm. Therefore we tried several different p_c-p_m-combinations on a few random graphs containing 50 nodes.[5]

Based on experimental results (not to go further into detail), we have chosen a population size of 30 individuals. The crossover rates we tested are 0, 0.2, 0.4, 0.6, and 0.8, while the mutation rates cover the range from 0 to 0.4 by a step size of 0.1. Altogether, the Evolutionary Algorithm ran with 24 various p_c-p_m-settings on every test graph. It stopped after 3000 fitness evaluations. For each combination we took the average fitness (buffer cost) over ten independent runs.

Exemplary, the results for a particular graph are visualized by the 3D plot in Figure 9; the results for the other random test graphs look similar.

Obviously, mutation is essential to this problem. Setting p_m to 0 leads to the worst results of all combinations of probabilities. If p_m is greater than 0, the obtained average buffer costs are significantly smaller—almost independently of the choice of p_c. As can be seen in Figure 10 this is due to premature convergence. The curve representing the performance for $p_c = 0.2$ and $p_m = 0$ goes horizontally after about 100 fitness evaluations. No new points in the search space are explored. As a consequence, the Monte Carlo optimization method, that simply generates random points in the search space and memorizes the best solution, might be a better approach to this problem. We investigate this issue in Section 6.

[5]Graphs consisting of less nodes are not very well suited to obtain reliable values for p_c and p_m, because the optimum is yet reached after a few generations, in most cases.

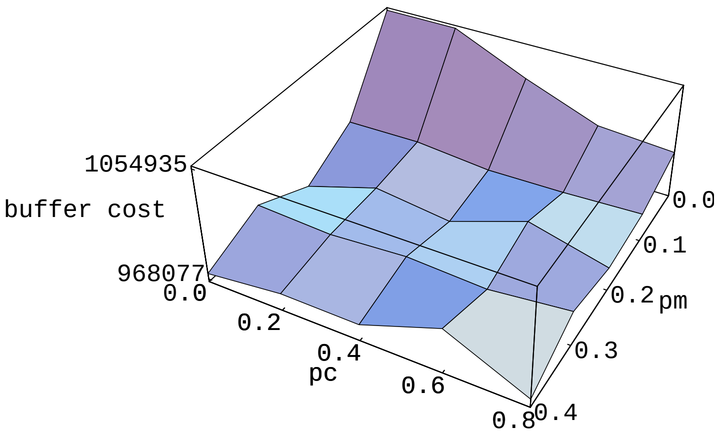


Figure 9: Influence of the crossover probability p_c and the mutation probability p_m on the average fitness for a particular test graph (3000 fitness evaluations).

On the other hand, the impact of the crossover operator on the overall performance is not as great as that of the mutation operator. With no mutation at all, increasing p_c yields decreased average buffer cost. But this is not the same to cases where $p_m > 0$. The curve for $p_c = 0.6$ and $p_m = 0.2$ in Figure 10 bears out this observation. Beyond it, for this particular test graph a mutation probability of $p_m = 0.2$ and a crossover probability of $p_c = 0$ leads to best performance. This might be interpreted as hint that *Hill Climbing* is also suitable in this domain. The Hill Climbing approach generates new points in the search space by applying a neighborhood function to the best point found so far. A comparison of the Evolutionary Algorithm to Hill Climbing is presented in Section 6.

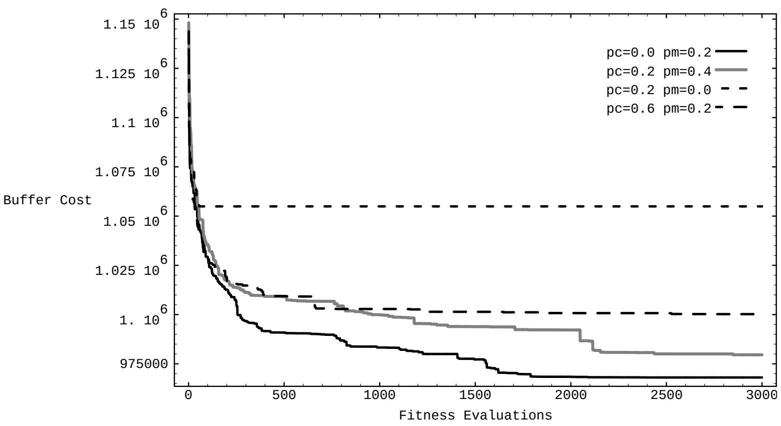


Figure 10: Performance of the Evolutionary Algorithm according to four different p_c-p_m-combinations; each graph represents the average of ten runs.

Nevertheless, with respect to the results on other test graphs, we found a crossover rate of $p_c = 0.2$ and a mutation rate of $p_m = 0.4$ to be most appropriate for this problem.

5 A randomized version of APGAN

The APGAN (Acyclic Pairwise Grouping of Adjacent Nodes) algorithm operates by repeatedly clustering pairs of actors to create a hierarchy of clusters, and then translating this cluster hierarchy into a hierarchy of nested loops. The cluster hierarchy is constructed by clustering exactly two adjacent actors at each step. At each clusterization step, a pair of *adjacent* actors is chosen that maximizes the cluster *repetition count* over all adjacent pairs that are *clusterable*, which means that they do not introduce directed cycles in the graph when clustered. The repetition count of a pair of actors X and Y in an SDF graph is given by $\gcd(q(X), q(Y))$, where $q()$ denotes the repetitions vector [6]. Two actors are adjacent in an SDF graph if an edge exists that connects the two actors. For full details on the APGAN algorithm, and the translation of cluster hierarchies into looped schedules we refer the reader to [7].

An inherent limitation in many direct optimization techniques such as APGAN is that such techniques cannot significantly exploit increased computational resources, nor the increased tolerance for high compilation times that developers of embedded systems perceive [23]. In contrast, Evolutionary Algorithms, and other probabilistic search methods have an unlimited capacity to absorb increases in the total computational power that can be expended on the optimization process.

However, it is usually possible to exploit potential nondeterminism within a direct optimization technique. For example, many greedy heuristics – like APGAN – operate by repeatedly examining a set of candidate choices (e.g. subgraphs to cluster, tasks to schedule next, etc.), and selecting a choice that maximizes some priority function over the set of candidates. Nondeterminism arises in such algorithms whenever there is more than one candidate that maximizes the priority function. Thus, it is often possible to augment greedy algorithms with probabilistic tie-breaking schemes to increase the search space covered by the optimization, and exploit increased computational capacity.

In addition to performing random tie breaking, which offers a limited form of nondeterminism, it is usually possible to randomize the whole greedy selection criteria. To perform a more thorough evaluation of our evolutionary strategy for memory compaction, we incorporated randomization into the APGAN heuristic, and compared the performance of the evolutionary strategy to our randomized version of APGAN (called RAPGAN).

As in APGAN, the RAPGAN algorithm repeatedly clusters pairs of adjacent actors. However, instead of always picking the adjacent pair that maximizes the repetition count, RAPGAN first sorts all clusterable adjacent pairs into a list L that is arranged in decreasing order of the cluster repetition count ($L[0]$, the first element of the list, is an adjacent pair that has the highest repetition count, and the repetition count of $L[i]$ is less than or equal to $L[j]$ whenever $i > j$). Then, the clustering

```
PROCEDURE RAPGAN
    IN:   p : {0, ··· , 1};
          graph : input_sdf_graph;
    OUT:  looped_schedule;
BEGIN
    current_graph := input_sdf_graph;
    WHILE current_graph contains more than one actor DO
        rank := -1;
        Form list L containing all clusterable adjacent actor pairs in
            current_graph;
        Sort L in decreasing order of repetition count (break ties arbitrarily);
        FOR (i=0; (i < N) AND rank < 0; i++) DO
            IF (random_nonneg(denominator(p)) < numerator(p)) THEN
            rank := i;
            FI
        OD
        IF (rank < 0) THEN
            rank := random_nonneg(N);
        FI
        new_graph := form_cluster(current_graph,L[rank]);
        current_graph := new_graph;
    OD
    Translate the cluster hierarchy represented by current_graph into a
        hierarchy of nested loops, and return the resulting looped schedule;
END
```

Figure 11: A pseudocode description of the RAPGAN algorithm.

candidate $L[0]$ is selected with probability "p," where "p" is the *randomization parameter* of the RAPGAN algorithm. If $L[0]$ is not selected, then $L[1]$ is selected with probability p, and so on until some $L[i]$ is selected or all elements of the list are exhausted without any selection made. In the latter case, a random integer R is obtained from a uniform distribution on $\{0, 1, ..., N-1\}$, where N is the number of elements in L, and $L[R]$ is selected for clustering.

Thus, at a given clustering step, the probability of choosing the mth element (clustering candidate) in the sorted list L for the clustering step is

$$p * ((1-p)^m) + ((1-p)^N)/N \quad \forall m = 0, 1, \cdots, N-1 \tag{8}$$

The case $p = 1$ corresponds to the original APGAN algorithm, and as p decreases from 1 to 0, repetition count has progressively less influence on clustering decisions. When $p = 0$, clustering is always performed by randomly sampling a uniform distribution over the current set of clusterable adjacent actor pairs.

Figure 11 shows a pseudo-code description of the RAPGAN algorithm.

Here p is a rational number in the range $0 \leq p \leq 1$ in reduced fraction form. The routine "random_nonneg(m)" returns a randomly-selected integer from a uniform distribution on $\{0, 1, ..., m\}$. Routine "form_cluster(g, z)" consolidates the adjacent pair z in g into a single hierarchical node, and returns the resulting graph.

6 Experiments

To evaluate the performance of the Evolutionary Algorithm we tested it on several practical examples of acyclic, multirate SDF graphs as well as on 200 acyclic random graphs, each containing 50 nodes and having 100 edges in average. The obtained results were compared against the outcomes produced by APGAN, RAPGAN, RPMC, Monte Carlo (MC), and Hill Climbing (HC). We also tried a slightly modified version of the Evolutionary Algorithm which first runs APGAN and then inserts the computed topological sort into the initial population.

Table 2 shows the results of applying GDPPO to the schedules generated by the various heuristics on several practical SDF graphs; the satellite receiver example is taken from [26], whereas the other examples are the same as considered in [6]. The probabilistic algorithms ran once on each graph and were aborted after 3000 fitness evaluations. Since the time needed by RAPGAN could not be measured in terms of fitness evaluations, several RAPGAN runs[6] were performed on each graph, so that the total of the run-time was equal to the Evolutionary Algorithm's run-time on that particular graph (3000 fitness evaluations); the best value achieved during the various runs was taken as the final result. Additionally, an exhaustive search with a maximum run-time of 1 hour was carried out; as it only completed in two cases[7], the search spaces of these problems seem to be rather complex.

In all of the practical benchmark examples that make up Table 2 the results achieved by the Evolutionary Algorithm equal or surpass the ones generated by RPMC. Compared to APGAN on these practical examples, the Evolutionary Algorithm is neither inferior nor superior; it shows both better and worse performance in two cases each. However, the randomized version of APGAN is only outperformed in one case by the EA. Furthermore, the performance of the Hill Climbing approach is almost identical to performance of the Evolutionary Algorithm. The Monte Carlo simulation, however, performs slightly worse than the other probabilistic approaches.

Although the results are nearly the same when considering only 1500 fitness evaluations, the Evolutionary Algorithm (as well as Monte Carlo and Hill Climbing) cannot compete with APGAN or RPMC concerning run-time performance. E.g., APGAN needs less than 2.3 second for all graphs on a SUN SPARC 20, while the run-time of the Evolutionary Algorithm varies from 0.1 seconds up to 5 minutes (3000 fitness evaluations).

The results concerning the random graphs are summarized in Table 3; again, the

[6]The randomization parameter p was set to 0.5.

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

System	BMLB	APGAN (RAPGAN)	RPMC	MC	HC	EA	EA + APGAN
Fractional decimation	47	47 (47)	52	47	47	47	47
Laplacian pyramid	95	99 (99)	99	99	99	99	99
Nonuniform filterbank (1/3, 2/3 splits) (4 channels)	85	137 (126)	128	143	126	126	126
Nonuniform filterbank (1/3, 2/3 splits) (6 channels)	224	756 (642)	589	807	570	570	570
QMF nonuniform-tree filterbank	154	160 (160)	171	165	160	160	159
QMF filterbank (one-sided tree)	102	108 (108)	110	110	108	108	108
QMF analysis only	35	35 (35)	35	35	35	35	35
QMF tree filterbank (4 channels)	46	46 (46)	55	46	47	46	46
QMF tree filterbank (8 channels)	78	78 (78)	87	78	80	80	78
QMF tree filterbank (16 channels)	166	166 (166)	200	188	190	197	166
satellite receiver	1540	1542 (1542)	2480	1542	1542	1542	1542

Table 2: Comparison of performance on practical examples; the probabilistic algorithms stopped after 3000 fitness evaluations.

stochastic approaches were aborted after 3000 fitness evaluations.[8] Interestingly, for these graphs APGAN only in 15% of all cases is better than Monte Carlo and only on in two cases better than the Evolutionary Algorithm. On the other hand, it is outperformed by the Evolutionary Algorithm 99% of the time.[9] This is almost identical to the comparison between Hill Climbing and APGAN. As RPMC is known to be better suited for irregular graphs than APGAN [6], its better performance (65.5%) is not surprising when directly compared to APGAN. Although, it is beaten by the Evolutionary Algorithm as well as Hill Climbing in 95.5% and 96.5% of the time, respectively. Also, RAPGAN outperforms APGAN and RPMC at a wide margin; compared to the Evolutionary Algorithm and Hill Climbing directly, it shows slightly worse performance. These results are very promising, but have to be

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

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

<	APGAN	RAPGAN	RPMC	MC	HC	EA	EA + APGAN
APGAN	0%	0.5%	34.5%	15%	0%	1%	0%
RAPGAN	99.5%	0%	94.5%	93.5%	15.5%	27.5%	21%
RPMC	65.5%	5.5%	0%	29.5%	3.5%	4.5%	2.5%
MC	85%	6.5%	70.5%	0%	0.5%	0.5%	1%
HC	100%	84.5%	96.5%	99.5%	0%	70%	57%
EA	99%	72%	95.5%	99.5%	22%	0%	39%
EA + APGAN	100%	78.5%	97.5%	99%	32.5%	53.5%	0%

Table 3: Comparison of performance on 200 50-actor SDF graphs (3000 fitness evaluations); for each row the numbers represent the fraction of random graphs on which the corresponding heuristic outperforms the other approaches.

considered in association with their quality, i.e., the magnitude of the buffer costs achieved.

Thus, we investigate the issue of deviation in the results on the random graphs in the following subsections. In Section 6.1 we analyze the outcomes produced by the Evolutionary Algorithm regarding RPMC. Section 6.2 compares the Evolutionary Algorithm to APGAN and examines, whether a combination of both approaches improves the overall performance; Section 6.3 extends the comparison the randomized APGAN. Finally, the remaining two subsections compare the Evolutionary Algorithm to the other probabilistic optimization methods, Monte Carlo and Hill Climbing, respectively.

6.1 Comparing the Evolutionary Algorithm to RPMC

Relative to APGAN, RPMC is expected to perform well on graphs that have rather irregular rate changes and irregular topologies [6]; although, it has no known optimality property relevant to practical graphs. Therefore, the question is whether RPMC or the Evolutionary Algorithm is the more appropriate complement to APGAN.

Table 3 shows that RPMC is outperformed by the Evolutionary Algorithm in 95.5% of the time (3000 fitness calculations[10]). Moreover, the buffer costs achieved by RPMC are significantly greater than the ones computed by the Evolutionary Algorithm. In average the results are worse by a factor of 1.58 and 1.6 in the case of 1500 and 3000 fitness evaluations respectively. In one case the Evolutionary Algorithm performs better by a factor of 10; on twelve random graphs this factor is greater than 3.

These results indicate that the Evolutionary Algorithm is superior to RPMC on highly irregular graphs. Additionally, it performs also better on the practical graphs listed in Table 2. Therefore, the Evolutionary Algorithm is the better choice when extra execution time is tolerable.

We also thought of incorporating RPMC into the Evolutionary Algorithm (similar to APGAN). However, since RPMC does not have any optimality property and

[10]94% in the case of 1500 fitness calculations.

surpasses the combination of Evolutionary Algorithm and APGAN in only 2.5% of the time (cf. Table 3), we did no further investigations in this direction. Nevertheless, one might combine APGAN, RPMC and Evolutionary Algorithm as the run-time of APGAN and RPMC can be neglected in comparison to the evolutionary approach.

6.2 Comparing the Evolutionary Algorithm to APGAN

Regarding highly irregular graphs, the Evolutionary Algorithm is superior to APGAN as the results presented in Table 3 indicate. The buffer costs achieved by the evolutionary approach are half the costs computed by APGAN in average. A deviation up to the factor 28 is achieved on a particular graph, in six cases an improvement by a factor greater than 5 can be observed. As stated in [7], APGAN performs well on graphs that have relatively regular topological structures and rate changes. Since large random graphs are rather expected to be irregular, this may explain the bad performance of APGAN relative to the Evolutionary Algorithm on our collection of random graphs.

Inserting the APGAN solution into the initial population seems to have only slight influence on the quality of results. After 3000 fitness calculations the average buffer cost are smaller by a factor of 1.0007 which is rather caused by random effects than due to better performance. On the other hand, regarding the case of 1500 fitness evaluations, the extension of the Evolutionary Algorithm leads to an improvement of a factor 1.039. This might indicate that this extension converges slightly faster towards the end result. We examined that issue on three random graphs, comparing the performance of both approaches in dependence on the number of fitness evaluations (see Figure 12 on page 32). With respect to one particular graph, this hypothesis could be confirmed, while on the other two graphs we found no significant differences in the behavior.

Nevertheless, the Evolutionary Algorithm and APGAN complement one another when regarding both regular and irregular graphs. On the one hand, APGAN exploits regularity that arises commonly in practical applications, and is provably optimal for an interesting class of graphs ([6]). On the other hand, the run-time needed by APGAN can be neglected in comparison to the Evolutionary Algorithm. Hence, it is a good idea to combine the two approaches in the presented manner. Table 3 shows that 53.5% of the time smaller buffer costs can be achieved, while the Evolutionary Algorithm without APGAN outperforms the combined approach in only 39% of all cases.

6.3 Comparing the Evolutionary Algorithm to the Randomized APGAN

In order to guarantee a fair comparison to the other stochastic algorithms, RAPGAN was applied to each of the considered graphs in the following manner:

1. Reset timer.

2. Execute RAPGAN and store result.

3. IF elapsed time > time needed by the Evolutionary Algorithm on that particular graph
   THEN go to step 4)
   ELSE go to step 2).

4. Print best result of all RAPGAN runs.

In contrast to the Evolutionary Algorithm, the run time of RAPGAN varied substantially on the random graphs. On particular graphs, RAPGAN was executed about 70 times, while on other graphs up to 900 runs were possible within the given time limit. Additionally, all experiments were performed with five different randomization parameters p: $0, \frac{1}{8}, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}$.

The results concerning the highly irregular graphs are summarized in Table 4. Regardless of the magnitude of the buffer costs achieved, the Evolutionary Algorithm can be said to be superior to RAPGAN. A closer look at the quality of the results, however, shows that the performance of RAPGAN is only slightly worse. In average the RAPGAN outcomes deviate only by a factor of 1.02 from the outcomes produced by the Evolutionary Algorithm. This can be interpreted as a hint that the Evolutionary Algorithm might be improved by incorporating domain knowledge (as it is the case for RAPGAN). Probably, most potentialities are related to the crossover operator, as the results presented in Section 4.5 together with the comparison to Hill Climbing (Section 6.5) indicate. More specialized crossover operators that make direct use of domain knowledge may speed and improve the evolution process. In this paper, however, we did not investigate this further.

When comparing the different values for p, it becomes obvious that randomization is the key factor for the improvement of APGAN. Even for $p = 0$ the performance of RAPGAN is close to the performance of the Evolutionary Algorithm. The results shown in Table 4 also point out the influence of the randomization parameter p. Increasing p leads to improvements on particular graphs (the numbers in the second row of Table 4 are increasing from left to right). On the other hand, this goes together with worse performance on other graphs, as the performance ratios in the last row in Table 4 are decreasing from right to left (exception: $p = 0$).

6.4 Evolutionary Algorithm versus Monte Carlo

The Monte Carlo implementation is very simple. Based an a uniform probability distribution, a certain number of topological sorts (bounded by the maximum number of fitness calculations) is generated randomly and to each of them GDPPO is applied. Finally, the topological sort with minimal buffer cost among all considered is the outcome of the algorithm. Since no mutation and crossover are necessary, this algorithm runs slightly faster than the Evolutionary Algorithm.

Disregarding the quality of results, the Evolutionary Algorithm clearly outperforms Monte Carlo (99% of the time, cf. Table 3), while there is only one case where Monte Carlo yields a better result. This is also true when considering the deviation

	RAPGAN $p = 0$	RAPGAN $p = \frac{1}{8}$	RAPGAN $p = \frac{1}{4}$	RAPGAN $p = \frac{1}{2}$	RAPGAN $p = \frac{3}{4}$
EA better	82% (78.5%)	73% (69.5%)	72% (66%)	72% (68.5%)	71.5% (69%)
EA worse	18% (21.5%)	26.5% (30.5%)	27.5% (34%)	27.5% (31.5%)	28.5% (30.5%)
$\frac{\text{RAPGAN}}{\text{EA}} \approx$	1.0273 (1.0301)	1.0082 (1.0023)	1.0129 (1.0111)	1.0155 (1.0099)	1.0210 (1.0138)

Table 4: Comparison of the Evolutionary Algorithm with the randomized APGAN; the latter ran with five different randomization parameters. The last row gives the average ratio of the RAPGAN buffer costs to the costs computed by the Evolutionary Algorithm. The numbers in parentheses refer to the experiments regarding 1500 fitness evaluations.

of results: In average, the buffer costs computed by the Evolutionary Algorithm are only a fraction of 0.84 of the costs produced by the Monte Carlo heuristic. The case of 1500 fitness evaluations looks identical.

These results are substantiated by the curves shown in Figure 12 on page 32. With it, we considered three random graphs and took the mean fitness value of ten independent runs at each point in time. Monte Carlo performs worse than the Evolutionary Algorithm as well as Hill Climbing.

In summary it may be said that the Evolutionary Algorithm is far better suited for this application domain than the Monte Carlo approach—in contrast to the presumption we made in Section 4.5 that the opposite may be true.

6.5 Evolutionary Algorithm versus Hill Climbing

We implemented Hill Climbing similarly to the Evolutionary Algorithm. First a starting point p, representing a permutation over the graph nodes, is chosen at random. Then the following loop is executed $n - 1$ times, where n denotes the maximum number of fitness evaluations:

1. A new point p' is created by applying the scramble list mutation operator to p (cf. Section 4.4).

2. Both p and p' are transformed into a topological sort by means of the repair mechanism described in Section 4.3.

3. The resulting topological sorts are evaluated by the SDF framework; if the phenotype of p' requires less buffer memory than the phenotype of p, p is set to p'.

Compared to the Evolutionary Algorithm, the results listed in Table 3 seem to indicate a superiority of the Hill Climbing method; its results are better 70% of the time and worse only 22% of the time. But when we examined the variations of the buffer costs computed by the two algorithms, we found no reliable indication for this hypothesis: In average, the performance of Hill Climbing is better by a factor of 1.0019—a non-significant margin.

Also the curves plotted in Figure 12 do not provide evidence of the superiority or the inferiority of one approach. Further investigations are necessary to answer this question.

7 Summary and Conclusions

We have proposed a new approach for finding buffer-optimal schedules among the set of program-memory schedules for uni-processor implementations of SDF graphs using a two-step approach: An Evolutionary Algorithm (EA) is used to explore the search space of legal actor firing orders. During the fitness computation, a $\mathcal{O}(N^3)$ dynamic programming post-optimizer calculates a loop nest with minimum buffer cost for the given actor firing order.

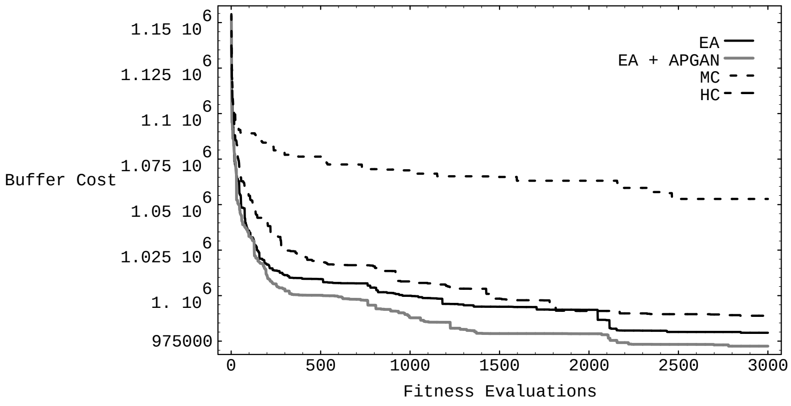
The results obtained have shown that this approach, though being more computationally expensive than existing algorithms such as APGAN and RPMC, promise to find better solutions in a reasonable amount of run-time. The differences in quality seem to become even more severe for larger graphs with irregular structures. Here, we tested the performance only for graphs with up to 50 nodes.

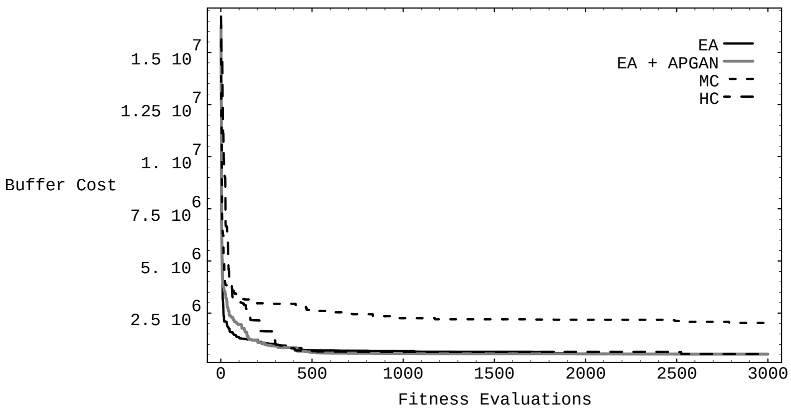
The experiments have also shown that Hill Climbing as well as the randomized APGAN could be good alternatives to the EA approach. Also Simulated Annealing and other stochastic optimization methods might be tried on this problem. However, in this paper, we did not investigate this further.

The ease to define and change the fitness function when using an Evolutionary Algorithm makes us think of the following extensions as part of the continuation of this work:

- *Exploitation of different buffer allocation schemes:*
  The buffer cost model used in our experiments so far considered only statically allocated contiguous memory segments that are not shared between different edge buffers. However, it should be possible to consider also memory-buffer sharing and more complex types of memory allocation. This should be possible be considering slightly more complex fitness functions. In this case, the optimization procedure does not only determine the optimum buffer costs but also makes decisions of what buffer organization to choose for each edge of a given graph.

- *Exploitation of optimal flat single appearance schedules:*
  The generated topological sorts could be considered directly as flat single appearance schedules. For these, the dynamic programming post-optimizer could be replaced by a memory allocation algorithm that exploits sharing between different edges. Because this problem is NP-hard, too (for non-unit buffer-sizes), we are currently developing efficient buffer allocation heuristics based on the life-time of memory blocks.





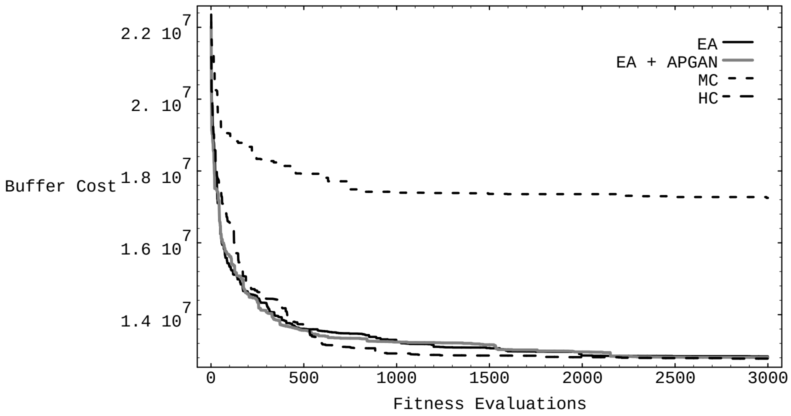


Figure 12: Performance of the probabilistic methods on three different random graphs. For both each graph and each method we took the average buffer cost over ten independent runs.

- *Consideration of general (non-acyclic) SDF graphs:*
  Here, we considered only acyclic SDF graphs. This had the advantage that the coding of actors had a linear space-complexity in terms of the number of nodes in the graph. This low complexity, however, was obtained at the cost of only exploiting buffer-memory schedules at the front of constant program-memory (single appearance schedules). Questions such as what is a minimal program-memory for designs at the front of buffer-memory optimal schedules could not be answered. Note, however, that in case of multiple-appearance schedules, the consideration of an amount of actor firings that is given by the number of entries in the minimal repetition vector seems to be unavoidable. Whether this approach is not prohibitive for typical DSP algorithms, however, will first have to be shown.

  Finally, not only these two criterion are of importance in the generation of DSP implementations of SDF graphs, but also other criterion such as execution time, and throughput, respectively: It would be interesting to investigate trade-offs between code-inlining and subprogram generation so to trade between program memory and execution time. In particular, code-inlining might still not be favorable in case of SOS (system on a chip) implementations of DSP algorithms. There, subprogram generation may lead to a dramatic decrease of required program memory in case multiple-appearance schedules are investigated. Whether EAs will help to answer these questions will be a longer term project.

We'd like to thank the anonymous reviewer of [27] for suggesting the development of a randomized version of APGAN or RPMC.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Principles of Compiler Design*. Addison-Wesley, Reading, MA, 1986.

[2] Thomas Bäck, Ulrich Hammel, and Hans-Paul Schwefel. Evolutionary computation: Comments om the history and current state. *IEEE Transactions on Evolutionary Computation*, 1(1):3–17, 1997.

[3] S. Bhattacharyya. Compiling data flow programs for digital signal processing. Technical Report UCB/ERL M94/52, Electronics Research Laboratory, UC Berkeley, July 1994.

[4] S. Bhattacharyya, J. T. Buck, S. Ha, and E. A. Lee. Generating compact code from dataflow specifications of multirate signal processing specifications. *IEEE Trans. on Circuits and Systems - I: Fundamental Theory and Applications*, 42(3):138–150, March 1995.

[5] S. Bhattacharyya and E. A. Lee. Looped schedules for dataflow descriptions of multirate signal processing algorithms. *Journal of Formal Methods in System Design*, 5(3), December 1994.

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

[7] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations. *J. Design Automation for Embedded Systems*, January 1997.

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

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

[10] Michael de la Maza and Bruce Tidor. An analysis of selection procedures with particular attention paid to proportional and boltzmann selection. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 124–131, San Mateo, California, 1993. Morgan Kaufmann.

[11] T. C. Denk and K. K. Parhi. Lower bounds on memory requirements for statically scheduled dsp programs. *J. of VLSI Signal Processing*, pages 247–264, 1996.

[12] J. Fabri. *Automatic Storage Optimization*. UMI Research Press, 1982.

[13] B. R. Fox and M. B. McMahon. Genetic operators for sequencing problems. In Gregory J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 284–300. Morgan Kaufmann, San Mateo, California, 1991.

[14] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, New York, 1979.

[15] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, Massachusetts, 1989.

[16] W.-C. Hsu. Register allocation and code scheduling for load/store architectures. Technical report, Department of Computer Science, University of Wisconsin at Madison, 1987.

[17] D. J. Kolson, A. N. Nicolau, N. Dutt, and K. Kennedy. Optimal register assignment to loops for embedded code generation. *ACM Trans. on Design Automation of Electronic Systems*, 1(2):251–279, 1996.

[18] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, Massachusetts, 1992.

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

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

[21] R. Leupers and P. Marwedel. Time-constrained code compaction for DSP's. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 5(1):112–122, March 1997.

[22] S. Y. Liao, S. Devadas, and K. Keutzer. Code density optimization for embedded DSP processors using data compression techniques. *IEEE Transactions on Computer-Aided Design*, 17(7):601–608, July 1998.

[23] P. Marwedel and G. Goossens (eds.). *Code generation for embedded processors.* Kluwer Academic Publishers, Norwell, MA, 1995.

[24] P. K. Murthy, S. S. Bhattacharyya, and E. A. Lee. Minimizing memory requirements for chain-structured Synchronous Data Flow Programs. In *Proc. of ICASSP-94*, Adelaide, Australia, 1994.

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

[26] S. Ritz, M. Willems, and H. Meyr. Scheduling for optimum data memory compaction in block diagram oriented software synthesis. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, volume 4, pages 2651–2654, May 1995.

[27] Jürgen Teich, Eckart Zitzler, and Shuvra S. Bhattacharyya. Buffer memory optimization in DSP applications — An evolutionary approach. In *Parallel Problem Solving from Nature (PPSN-V)*, pages 885–894, Amsterdam, September 1998. Springer Lecture Notes in Computer Science (LNCS) 1498.

[28] Jürgen Teich, Eckart Zitzler, and Shuvra S. Bhattacharyya. Optimized software synthesis for digital signal processing algorithms - an evolutionary approach. In *IEEE Workshop on SIGNAL PROCESSING SYSTEMS (SiPS)*, pages 589–598, Boston, MA, October 1998.