# Evolutionary Algorithms for the Synthesis of Embedded Software

Eckart Zitzler, Jürgen Teich, and Shuvra S. Bhattacharyya

*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 [1], 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.**

*Keywords*— **Dataflow, evolutionary algorithms, memory management, software synthesis.**

## I. Introduction

SOFTWARE synthesis has become an important component of the implementation process for embedded VLSI systems due to flexibility and time-to-market considerations.

Synchronous dataflow (SDF) [1] 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. The SDF model is used in industrial DSP design tools, e.g., SPW by Cadence, COSSAP by Synopsys, as well as in research-oriented environments, e.g., Ptolemy [2], GRAPE [3], and COSSAP [4]. 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 by code inlining. Subroutine calls may have unacceptable overhead, especially if there are many small tasks.

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, and so called *single appearance schedules* where each actor appears only once are program memory optimal. Results on the existence of such schedules for general SDF graphs are discussed in [5].

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 [5] that considers general SDF graphs and generates schedules for acyclic subgraphs using our approach.

E. Zitzler was supported in this work by the Swiss National Science Foundation; S. S. Bhattacharyya was supported by the US National Science Foundation (CAREER, MIP9734275).

Eckart Zitzler is with the Computer Engineering and Networks Laboratory (TIK), ETH Zurich, Switzerland. Jürgen Teich is with the Computer Engineering Laboratory (DATE), University of Paderborn, Germany. Shuvra S. Bhattacharyya is with the Department of Electrical and Computer Engineering, and the Institute for Advanced Computer Studies, University of Maryland at College Park.

### A. Motivation

For a given acyclic SDF graph, 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. 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 [5], a heuristic called APGAN is introduced that is used inside state of the art design systems such as Ptolemy and 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 and providing good performance on many applications, these heuristics can sometimes produce results that are far from optimal [6]. However, in embedded DSP applications, code-optimality is often critical, and implementations are changed very infrequently, if at all [7]. Hence, compilation times in the order of minutes, hours, or even days are tolerable. Thus, the motivation of the following work was to develop a methodology that can exploit increased tolerance for long compile time to improve significantly on results produced by state of the art algorithms such as APGAN and RPMC.

### B. Proposed Approach

We propose 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 of a given SDF graph.
- For each topological sort, a buffer optimal schedule is constructed based on a well-known dynamic programming post optimization step, called GDPPO [5]. Given a lexical ordering $L$ of actors of an SDF graph $G$, GDPPO constructs a schedule whose buffer memory requirement is less than or equal to the lowest buffer memory requirement over all single-appearance schedules for $G$ that have lexical ordering $L$. If $G$ is without delay, then the output of GDPPO will leave the lexical ordering $L$ unchanged; otherwise, delays in $G$ may be exploited to improve the lexical ordering. The run-time of GDPPO is $\mathcal{O}(N^3)$, where $N$ is the number of actors.

### C. Motivation for Using an Evolutionary Algorithm (EA)

In order to efficiently explore the search space, an evolutionary algorithm is a promising choice for the following reasons:

• Topological sorts may be easily coded using an evolutionary algorithm. Details on the coding scheme will be given in the following section.
• 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.

### D. Related Work

The interaction between instruction scheduling and register allocation in procedural language compilers has been studied extensively [8], [9], and optimal management of this interaction is known to be intractable [10]. 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 [7], [12]. 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 [13], [14]. 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.

Preliminary ideas of our work can be found in [15], [16], and a detailed presentation of the ideas in this paper can be found in [6].

## II. The SDF-scheduling framework

### A. Background and Notation

*Definition II.1* (SDF graph) An SDF graph $G$ denotes a 5-tuple $G = (V, A, \mathit{produced}, \mathit{consumed}, \mathit{delay})$ where
• $V$ is the set of nodes (*actors*) ($V = \{v_1, v_2, \cdots, v_N\}$).
• $A$ is the set of directed edges. With $\mathit{source}(\alpha)$ ($\mathit{sink}(\alpha)$), we denote the source node (target node) of an edge $\alpha \in A$.
• $\mathit{produced} : A \to \mathbf{N}$ denotes a function that assigns to each directed edge $\alpha \in A$ the number of produced tokens $\mathit{produced}(\alpha)$ per invocation of actor $\mathit{source}(\alpha)$.
• $\mathit{consumed} : A \to \mathbf{N}$ denotes a function that assigns to each directed edge $\alpha \in A$ the number of consumed tokens per invocation of actor $\mathit{sink}(\alpha)$.
• $\mathit{delay} : A \to \mathbf{N}_0$ denotes the function that assigns to each edge $\alpha \in A$ the number of initial tokens $\mathit{delay}(\alpha)$.

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 queued on 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*.

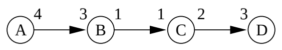

Fig. 1. A simple SDF graph.

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 [1]. 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 II.1:* Figure 1 shows an example of an SDF graph with four nodes and three edges. The four nodes (actors) are labeled $A, B, C, D$. 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

$$(\infty(2ABC)DABCDBC(2ABCD)A(2BC)(2ABC)A(2BCD)) \quad (1)$$

represents a valid schedule for this graph.
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 a *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 II.2:* The schedules $(\infty(9A)(12B)(12C)(8D))$ and $(\infty(3(3A)(4BC))(8D))$ are both valid single appearance schedules for the graph shown in Fig. 1.

### B. Code generation model and objective function

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

Implied by this model of code generation, any valid single appearance schedule gives the minimum code space (program memory) cost. This approximation, however, neglects the code size overhead of implementing loops.[1]

The objective function to minimize is the buffer cost (data memory) $c$ which for a given schedule $S$ is given by

$$c = c(S) = \sum_{\alpha \in A} \mathit{max\_tokens}(\alpha, S)$$

[1]In practical SDF models of applications, the code size overhead of a loop is typically small compared to the size of individual code blocks. Thus, in our context, neglecting loop overhead does not lead to misleading results in focusing on single appearance schedules for code size minimization.

Here, $max\_tokens(\alpha, S)$ denotes the maximum number of tokens that accumulate on edge $\alpha$ during the execution of $S$.[2]

## III. The Evolutionary Algorithm

The term *evolutionary algorithm (EA)* [17] stands for a class of computational models that mimic natural evolution in order to solve arbitrary optimization problems. In general, an EA is characterized by three facts: i) a set (*population*) of solution candidates (*individuals*) is maintained, which ii) undergoes a *fitness*-based selection process and iii) is manipulated by genetic operators, usually recombination and mutation.

Here, each individual is a permutation over the set of nodes in the given SDF graph. Since we are only interested in topological sorts, each permutation is deterministically transformed into a topological sort using a simple repair mechanism. Furthermore, two special order-based genetic operators were chosen that do not destroy the permutation property of individuals. In detail, the flow of the EA implemented in this study is as follows.

*Step 1:* **Initialization**: Create an initial population $P$ containing $N$ randomly generated permutations over the set of nodes.

*Step 2:* **Fitness assignment**: For each individual (permutation) $i \in P$ do

1. Map the permutation $i$ unambiguously to a topological sort $T$ by running a topological sort algorithm on $i$—here, the tie between several nodes with no incoming edges is not broken at random, but always the leftmost node in $i$ is selected.
2. Run GDPPO on $T$, yielding buffer cost $c$.
3. Assign $i$ the fitness value $f_i = c$.

*Step 3:* **Selection**: Reproduce individuals using binary tournament selection, i.e, set $P' = \emptyset$, and perform the following two steps $N$ times:

1. Select two individuals $i, j \in P$ at random.
2. If $f_i < f_j$ then copy $i$ to $P'$ else copy $j$ to $P'$.

*Step 4:* **Recombination**: Set $P'' = \emptyset$. While $P'$ not empty do

1. Choose two individuals $i, j \in P'$ at random and remove them from $P'$.
2. Recombine $i$ and $j$ using uniform order-based crossover [18]. The resulting children are $k$ and $l$.
3. Add $k$ and $l$ to $P''$ with probability $p_c$ (crossover rate). Otherwise add $i$ and $j$ to $P''$.

*Step 5:* **Mutation**: Set $P''' = \emptyset$. For each individual $i \in P''$ do

1. Mutate $i$ by permuting the nodes between two selected positions (scramble sublist mutation [18]). The resulting mutated individual is $j$.
2. Add $j$ to $P'''$ with probability $p_m$ (mutation rate). Otherwise add $i$ to $P'''$.

*Step 6:* **Elitism**: Replace one arbitrary individual in $P'''$ by $i \in P$ for which $f_i$ is minimal regarding $P$.

*Step 7:* **Termination**: Set $P = P'''$. If the maximum number of fitness evaluations $F_{max}$ is reached then stop else go to Step 2.

[2]This model assumes that a distinct area of memory is allocated for each edge in the graph.

TABLE I
Comparison of buffer cost on the practical examples.[3]

| Ex. | APGAN (RAPGAN) | RPMC | MC | HC | EA | EA + APGAN |
|---|---|---|---|---|---|---|
| 1 | 47 (47) | 52 | 47 | 47 | 47 | 47 |
| 2 | 99 (99) | 99 | 99 | 99 | 99 | 99 |
| 3 | 137 (126) | 128 | 143 | 126 | 126 | 126 |
| 4 | 756 (642) | 589 | 807 | 570 | 570 | 570 |
| 5 | 160 (160) | 171 | 165 | 160 | 160 | 159 |
| 6 | 108 (108) | 110 | 110 | 108 | 108 | 108 |
| 7 | 35 (35) | 35 | 35 | 35 | 35 | 35 |
| 8 | 46 (46) | 55 | 46 | 47 | 46 | 46 |
| 9 | 78 (78) | 87 | 78 | 80 | 80 | 78 |
| 10 | 166 (166) | 200 | 188 | 190 | 197 | 166 |
| 11 | 1542 (1542) | 2480 | 1542 | 1542 | 1542 | 1542 |

## IV. A randomized version of APGAN

It is usually possible to systematically incorporate randomization into a deterministic optimization algorithm. This can significantly increase the search space covered by the algorithm, and exploit increased computational capacity. Here, we also compared the performance of the evolutionary strategy to a randomized version of APGAN (called RAPGAN). One key input to RAPGAN is a *randomization parameter* $p$, which determines the "degree of randomness" that is introduced into the APGAN scheduling approach. The case $p = 1$ corresponds to the original APGAN algorithm, and as $p$ decreases from 1 to 0, the deterministic factors that guide the APGAN algorithm have progressively less influence on scheduling decisions. Details of the RAPGAN algorithm formulation can be found in [6].

## V. Experiments

### A. Methodology

The EA was tested 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). MC generates a certain number of solutions at random, the best topological sort found is taken as the outcome. In contrast, HC generates new points in the search space by mutating the best topological sort found so far. We also tried a slightly modified version of the EA that first runs APGAN and then inserts the computed topological sort into the initial population (EA + APGAN).

Each EA run was performed using the parameters: $N = 30$, $p_c = 0.2$, $p_m = 0.4$. The EA, MC and HC ran for 3000 fitness evaluations each. In the case of RAPGAN, several runs were carried out per graph, such that the total of the run-time was equal to the EA's run-time on that particular graph; the best value achieved during the various runs was taken as the final result. The randomization parameter $p$ of RAPGAN was set to $p = 0.5$.

### B. Practical Examples of SDF Graphs

In all of the practical benchmark examples that make up Table I the results achieved by the EA equal or surpass

[3]The following systems were considered: 1) fractional decimation; 2) Laplacian pyramid; 3) nonuniform filterbank (1/3, 2/3 splits, 4 channels); 4) nonuniform filterbank (1/3, 2/3 splits, 6 channels); 5) QMF nonuniform-tree filterbank; 6) QMF filterbank (one-sided tree); 7) QMF analysis only; 8) QMF tree filterbank (4 channels); 9) QMF tree filterbank (8 channels); 10) QMF tree filterbank (16 channels); 11) satellite receiver.

TABLE II

Comparison of performance on 200 50-actor SDF graphs; for each row the numbers represent the fraction of random graphs on which the corresponding heuristic outperforms the other approaches.

| < | 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% |

the ones generated by RPMC. Compared to APGAN on these practical examples, the EA 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, HC and EA show almost identical performance, while MC achieves slightly worse results than the other probabilistic algorithms.

### C. Random SDF Graphs

The experiments concerning the random graphs are summarized in Table II.[4] Interestingly, for these graphs APGAN is better than MC only in 15% of all cases and better than the EA only in two cases. However, it is outperformed by the EA 99% of the time. This is almost identical to the comparison of HC and APGAN. As RPMC is known to be better suited for irregular graphs than APGAN [5], its better performance (65.5%) is not surprising when directly compared to APGAN. Nevertheless, it is beaten by the EA as well as HC in more than 95% of the time. Also, RAPGAN outperforms APGAN and RPMC by a wide margin; compared to both EA and HC directly, it shows slightly worse performance.[5]

In average the buffer costs achieved by the EA are half the costs computed by APGAN and only a fraction of 63% of the RPMC outcomes. Moreover, an improvement by a factor 28 can be observed on a particular random graph with respect to APGAN (factor of 10 regarding RPMC). Compared to MC, it is the same, although the margin is smaller (in average the results of the EA are a fraction of 0.84% of the costs achieved by MC). HC, however, might be an alternative to the EA, although the costs achieved by the EA deviate from the costs produced by HC only by a factor of 0.19% in average. This also suggests that i) the EA might be improved using a more specialized crossover operator and ii) Simulated Annealing, for instance, could be a good optimization algorithm with this problem, too. However, this has not been investigated further in this paper. Finally, the RAPGAN results are worse by less than 3% of the EA results with regard to the magnitude of the buffer cost.

## VI. Summary and Conclusions

Although being computationally more expensive than existing state of the art algorithms, our new approach for finding buffer-optimal schedules among the set of program-memory schedules for uni-processor implementations of SDF graphs using an evolutionary algorithm (EA) has been shown 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.

[4]The EA ran about 9 minutes on each graph, the time for running APGAN was consistently less than 3 seconds on a SUN SPARC 20.

[5]This holds for different values of the randomization parameter $p$ as has been verified experimentally.

## References

[1] E. Lee and D. Messerschmitt, "Synchronous dataflow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

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

[3] R. Lauwereins, M. Engels, J. A. Peperstraete, E. Steegmans, and J. V. Ginderdeuren, "Grape: A CASE tool for digital signal parallel processing," *IEEE ASSP Magazine*, vol. 7, no. 2, pp. 32–43, Apr. 1990.

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

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

[6] E. Zitzler, J. Teich, and S. S. Bhattacharyya, "Optimized software synthesis for DSP using randomization techniques (revised version of TIK Report 32).," Tech. Rep. 75, Institute TIK, ETH Zurich, Gloriastrasse 35, CH-8092 Zurich, Switzerland, 1999.

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

[8] W.-C. Hsu, "Register allocation and code scheduling for load/store architectures," Tech. Rep., Department of Computer Science, University of Wisconsin at Madison, 1987.

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

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

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

[12] 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*, vol. 1, no. 2, pp. 251–279, 1996.

[13] R. Leupers and P. Marwedel, "Time-constrained code compaction for DSP's," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 5, no. 1, pp. 112–122, March 1997.

[14] 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*, vol. 17, no. 7, pp. 601–608, July 1998.

[15] J. Teich, E. Zitzler, and S. S. Bhattacharyya, "Buffer memory optimization in DSP applications — An evolutionary approach," in *Parallel Problem Solving from Nature (PPSN-V)*, Amsterdam, Sept. 1998, pp. 885–894, Springer LNCS series vol. 1498.

[16] J. Teich, E. Zitzler, and S. S. Bhattacharyya, "Optimized software synthesis for digital signal processing algorithms - an evolutionary approach," in *IEEE Workshop on Signal Processing Systems (SiPS)*, Boston, MA, Oct. 1998, pp. 589–598.

[17] T. Bäck, U. Hammel, and H.-P. Schwefel, "Evolutionary computation: Comments om the history and current state," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 3–17, 1997.

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