Technical Report #UMIACS-TR-99-32, Institute for Advanced Computer Studies, University of Maryland, College Park, 20742, June, 1999.

Shared Memory Implementations of Synchronous Dataflow Specifications Using Lifetime Analysis Techniques

Praveen K. Murthy
Angeles Design Systems
pmurthy@angeles.com

Shuvra S. Bhattacharyya
ECE Dept., and Institute for Advanced Computer Studies, University of Maryland, College Park
ssb@eng.umd.edu

Abstract

There has been a proliferation of block-diagram environments for specifying and prototyping DSP systems. These include tools from academia like Ptolemy [6], and commercial tools like SPW from Cadence Design Systems, and Cossap from Synopsys. The block diagram languages used in these environments are usually based on dataflow semantics because various subsets of dataflow have proven to be good matches for expressing and modeling signal processing systems. In particular, synchronous dataflow (SDF)[14] has been found to be a particularly good match for expressing multirate signal processing systems [5].

One of the key problems that arises during synthesis from an SDF specification is scheduling. Past work on scheduling [3] from SDF has focused on optimization of program memory and buffer memory. However, in [3], no attempt was made for overlaying or sharing buffers. In this paper, we formally tackle the problem of generating optimally compact schedules for SDF graphs, that also attempt to minimize buffering memory under the assumption that buffers will be shared. This will result in schedules whose data memory usage is drastically lower than methods in the past have achieved. The method we use is that of lifetime analysis; we develop a model for buffer lifetimes in SDF graphs, and develop scheduling algorithms that attempt to generate schedules that minimize the maximum number of live tokens under the particular buffer lifetime model. We develop several efficient algorithms for extracting the relevant lifetimes from the SDF schedule. We then use the firstfit heuristic for packing arrays efficiently into memory. We report extensive experimental results on applying these techniques to several practical SDF systems, and show improvements that average 50% over previous techniques, with some systems exhibiting upto an 83% improvement over previous techniques.

1 Introduction

Block diagram environments are proving to be increasingly popular for developing DSP systems. The reasons for their popularity are many: block-diagram languages are visual and hence intuitive to use for engineers naturally used to conceptualizing systems as block diagrams, block diagram languages promote software reuse by encapsulating designs as modular, reusable components, and finally, these languages can be based on models of computation that have strong formal properties, enabling easier and faster development of bug-free programs. Block-diagram specifications also have the desirable property of not overspecifiying systems; this can enable a synthesis tool to exploit all of the concurrency and parallelism available at the system level.

In a block-diagram environment, the user connects up various blocks drawn from a library to form the system of interest. For simulation, these blocks are typically written in a high level language like C++. For software synthesis, the technique typically used is that of threading: a schedule is generated, and the code generator steps through this schedule and substitutes the code for each actor that it encounters in the schedule. The code for the actor may be of two types: it may be the HLL code itself, obtained from the actor in the simulation library. The overall code may now be compiled for the appropriate target. However, this is not as efficient as having a separate library of actors containing hand-optimized code targeted for software synthesis. For programmable DSPs, this means that the actors implement their functionality through hand-optimized assembly language segments. The code-generator, after stitching together the code for the entire system then simply assembles it and the resulting machine code can be run on the DSP.

For hardware synthesis, a similar approach can be taken, with blocks implementing their functionality in a hardware description language, like behavioral VHDL [11][27]. The generated VHDL description can then be used by a behavioral compiler to generate an RTL description of the system, that can be further compiled into hardware using design compilers and layout tools.

High-level language compilers for DSPs have been woefully inadequate in the past [28]. This has been because of the highly irregular architecture that many DSPs have, the specialized addressing modes such as modulo addressing, bit-reversed addressing, and small number of special purpose registers. Traditional compilers are unable to generate efficient code for such processors. This situation might change in the future, if DSP architectures converge to general-purpose architectures; for example, the C6 DSP from TI, the newest DSP architecture from TI, is a VLIW architecture and has a fairly good compiler. Even so, because of low power requirements, and cost constraints, the fixed-point DSP with the irregular architecture is likely to dominate in embedded applications for the foreseeable future. Because of the shortcomings of existing compilers for such DSPs, a considerable research effort has been undertaken to design better compilers for fixed point DSPs [18][16].

Synthesis from block diagrams is useful and necessary when the block diagram becomes the abstract specification rather than C code. Block diagrams also enable coarse grain optimizations based on knowledge of the restricted underlying models of computation; these optimizations are frequently difficult to perform for a traditional compiler. Since the first step in block diagram synthesis flows is the scheduling of the block diagram, we consider in this paper scheduling strategies for minimizing memory usage. Since the scheduling techniques we develop operate on the coarse-grain, system level description, these techniques are somewhat orthogonal to the optimizations that might be employed by tools lower in the flow. For example, a behavioral compiler has a limited view of the code, often confined to basic blocks within each block it is optimizing, and cannot make use of the global control and dataflow that our scheduler can exploit. Similarly, a high-level language compiler is also usually only good at optimizing basic blocks (where we use this term in a compiler sense), and does not have the global information that our scheduler does. The techniques we develop in this paper are thus complementary to the work that is being done on developing better compilers for DSPs [16][17][18]. In particular, the techniques we develop operate on the graphs at a high enough level that particular architectural features of the target processor are largely irrelevant. We assume that the actor library that the code generator has access to consists of either hand-optimized assembly code, or of specifications in a high-level language like C. If the latter, then we would have to invoke a C compiler after performing the SDF optimizations and threading the code together. Even though this might seemingly defeat the purpose of producing efficient code, since we are using a C compiler for a DSP (the compiler might not be very good as mentioned), studies have shown that for larger systems, C code produced this way compiles better than hand-written C for the entire system [8].

As already mentioned, dataflow is a natural model of computation to use as the underlying model for a block-diagram language for designing DSP systems. The blocks in the language correspond to actors in a dataflow graph, and the connections correspond to directed edges between the actors. These edges not only represent communication channels, conceptually implemented as FIFO queues, but also establish precedence constraints. An actor fires in a dataflow graph by removing tokens from its input edges and producing tokens on its output edges. The stream of tokens produced this way corresponds naturally to a discrete time signal in a DSP system. In this paper, we consider a subset of dataflow called synchronous dataflow (SDF) [14]. In SDF, each actor produces and consumes a fixed number of tokens, and these numbers are known at compile time. In addition, each edge has a fixed initial number of tokens, called delays.

2 Notation and background

Fig. 1(a) shows a simple SDF graph. Each edge is annotated with the number of tokens produced (consumed) by its source (sink) actor, and the "D" on the edge from actor A to actor B specifies a unit delay. Each unit of delay is implemented as an initial token on the edge. Given an SDF edge e, we denote the source actor, sink actor, and delay of e by $src(e)$, $snk(e)$, and $del(e)$. Also, $prod(e)$ and $cns(e)$

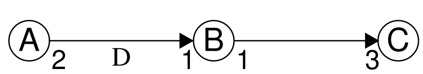


Fig 1. An example of an SDF graph.

denote the number of tokens produced onto e by $src(e)$ and consumed from e by $snk(e)$. An SDF graph is called **homogenous** if $src(e) = snk(e)$ for all edges e.

A **schedule** is a sequence of actor firings. We compile an SDF graph by first constructing a **valid schedule** — a finite schedule that fires each actor at least once, does not deadlock, and produces no net change in the number of tokens queued on each edge. Corresponding to each actor in the schedule, we instantiate a code block that is obtained from a library of predefined actors. The resulting sequence of code blocks is encapsulated within an infinite loop to generate a software implementation of the SDF graph.

SDF graphs for which valid schedules exist are called **consistent** SDF graphs. In [3], efficient algorithms are presented to determine whether or not a given SDF graph is consistent, and to determine the minimum number of times that each actor must be fired in a valid schedule. We represent these minimum numbers of firings by a vector q_G, indexed by the actors in G (we often suppress the subscript if G is understood). These minimum numbers of firings can be derived by finding the minimum positive integer solution to the **balance equations** for G, which specify that q must satisfy

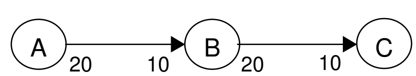
$$prod(e)q(src(e)) = cns(e)q(snk(e)) \text{, for every edge } e \text{ in } G.$$

The vector q, when it exists, is called the **repetitions vector** of G.

3 Constructing memory-efficient loop structures

In [3], the concept and motivation behind **single appearance schedules** (**SAS**) has been defined and shown to yield an optimally compact inline implementation of an SDF graph with regard to code size (neglecting the code size overhead associated with the loop control). An SAS is one where each actor appears only once when loop notation is used. Figure 2 shows an SDF graph, and valid schedules for it. The notation $2B$ represents the firing sequence BB. Similarly, $2(B(2C))$ represents the schedule loop with firing sequence $BCCBCC$. Schedules 2 and 3 in figure 2 are single appearance schedules since actors A, B, C appear only once. An SAS like the third one in Figure 2(b) is called **flat** since it does not have any nested loops. In general, there can be exponentially many ways of nesting loops in a flat SAS.

(a)



(b)

Valid Schedules

(1): ABCBCCC (2): A(2 (B(2 C)))

(3): A(2 B)(4 C) (4): A(2 (BC))(2

Fig 2. An example used to illustrate the interaction between scheduling SDF graphs and the memory requirements of the generated code.

Scheduling can also have a significant impact on the amount of memory required to implement the buffers on the edges in an SDF graph. For example, in Figure 2(b), the buffering requirements for the four schedules, assuming that one separate buffer is implemented for each edge, are 50, 40, 60, and 50 respectively.

4 Optimizing for buffer memory

We give priority to code-size minimization over buffer memory minimization; justification for this may be found in [3][19]. Hence, the problem we tackle is one of finding buffer-memory-optimal SASs, since this will give us the best schedule in terms of buffer-memory consumption amongst the schedules that have minimum code size.

In [19] and [3], we developed a post-processing technique based on dynamic programming (called DPPO), that generates an optimal loop hierarchy for any given SAS. The cost metric used for this approach is that each edge is implemented as a separate buffer. We briefly review that technique here.

Given an SDF graph G, a valid schedule S, and an edge e in G, $max\_tokens(e, S)$ denotes the maximum number of tokens that are queued on e during an execution of S. For example, if for fig. 1, $S_1 = (3A)(6B)(2C)$ and $S_2 = (3A(2B))(2C)$, then $(max\_tokens((A, B), S_1)) = 7$ and $(max\_tokens((A, B), S_2)) = 3$. We define the **buffer memory requirement** of a schedule S by

$$bufmem(S) \equiv \sum max\_tokens(e, S), \qquad \textbf{(EQ 1)}$$

where the summation is over all edges in G. Thus, $bufmem(S_1) = 7 + 6 = 13$, and $bufmem(S_2) = 3 + 6 = 9$.

The **lexical ordering** of a SAS S, denoted $lexorder(S)$, is the sequence of actors $(A_1, A_2, \ldots, A_n)$ such that each A_i is preceded lexically by $A_1, A_2, \ldots, A_{i-1}$. Thus, $lexorder((2(3B)(5C))(7A)) = (B, C, A)$. Given an SDF graph, an **order-optimal schedule** is a SAS that has minimum buffer memory requirement from among the valid SASs that have a given lexical ordering.

Suppose that G is a connected, delayless, acyclic SDF graph, S is valid SAS for G, $lexorder(S) = (A_1, A_2, \ldots, A_n)$, and S_{oo} is an order-optimal schedule for $(G, lexorder(S))$. If G contains at least two actors, then it can be shown [19] that there exists a valid schedule of the form $S_R = (i_L B_L)(i_R B_R)$ such that $bufmem(S_R) = bufmem(S_{oo})$ and for some $p \in \{1, 2, \ldots, (n-1)\}$, $lexorder(B_L) = (A_1, A_2 \ldots, A_p)$ and $lexorder(B_R) = (A_{p+1}, A_{p+2}, \ldots, A_n)$. Furthermore, from the order-optimality of S_{oo}, clearly, $(i_L B_L)$ and $(i_R B_R)$ must also be order-optimal.

From this observation, we can efficiently compute an order-optimal schedule for G if we are given an order-optimal schedule $S_{a,b}$ for the subgraph corresponding to each proper subsequence $A_a, A_{a+1}, \ldots, A_b$ of $(lexorder(S))$ such that (1). $(b-a) \le (n-2)$ *and* (2). $a = 1$ or $b = n$. Given these schedules, an order-optimal schedule for G can be derived from a value of x, $1 \le x < n$ that minimizes

$$bufmem(S_{1,x}) + bufmem(S_{x+1,n}) + \sum_{e \in E_s} TNSE(e), \text{ where}$$

$$E_s = \{e | ((src(e) \in \{A_1, A_2, \ldots, A_x\}) AND (snk(e) \in \{A_{x+1}, A_{x+2}, \ldots, A_n\}))\}$$

is the set of edges that "cross the split" if the schedule parenthesization is split between A_x and A_{x+1}.

DPPO is based on repeatedly applying this idea in a bottom-up fashion to the given lexical ordering $(lexorder(S))$. First, all two actor subsequences (A_1, A_2), (A_2, A_3), $\ldots$, (A_{n-1}, A_n) are examined and the minimum buffer memory requirements for the edges contained in each subsequence are recorded. This information is then used to determine an optimal parenthesization split and the minimum buffer memory requirement for each three actor subsequence (A_i, A_{i+1}, A_{i+2}); the minimum requirements for the two- and three-actor subsequences are used to determine the optimal split and minimum buffer memory requirement for each four actor subsequence; and so on, until an optimal split is derived for the original n-actor sequence $(lexorder(S))$. An order-optimal schedule can easily be constructed from a recursive, top-down traversal of the optimal splits [19].

In the rth iteration of this bottom up approach, we have available the minimum buffer memory requirement $b[p, q]$ for each subsequence $(A_p, A_{p+1}, \ldots, A_q)$ that has less than or equal to r members. To compute the minimum buffer memory requirement $b[i, j]$ associated with an $r+1$-actor subchain $(A_i, A_{i+1}, \ldots, A_j)$, we determine a value of $k \in \{i, i+1, \ldots, j-1\}$ that minimizes

$$b[i, k] + b[k+1, j] + c_{i,j}[k], \qquad \textbf{(EQ 2)}$$

where $b[x, x] = 0$ for all x and $c_{i,j}[k]$, the memory cost at the split if we split the subsequence between A_k and A_{k+1} is given by [19]

$$c_{i,j}[k] = \left(\sum_{e \in E_s} TNSE(e)\right) / (GCD(\{q_G(A_x) | (i \le x \le j)\})), \qquad \textbf{(EQ 3)}$$

where *gcd* denotes the greatest common divisor, and

$$E_s = \{e | ((src(e) \in \{A_i, A_{i+1}, \ldots, A_k\}) AND (snk(e) \in \{A_{k+1}, A_{k+2}, \ldots, A_j\}))\} \qquad \textbf{(EQ 4)}$$

is the set of edges that cross the split.

5 The shared buffer model

In this report, we develop scheduling techniques for minimizing buffer memory under a shared buffer memory model. Buffer sharing for looped schedules can be done at many different levels of "granularity". At the finest level of granularity, we can model the buffer on the edge as it grows over the execution of the loop, and then falls as the sink node on that edge consumes the data. The maximum number of live tokens would give the lower bound on how much memory would be required. An alternative model would be at the coarsest level, where we assume that once the source actor for an edge e starts writing tokens, $TNSE(e)$ tokens immediately become live, and stay live until the number of tokens on the edge becomes zero. In other words, even if there is one live token on the edge, we assume that an array of size $TNSE(e)$ has to be allocated and maintained until there are no live tokens. Figure 3 shows these two alternatives pictorially. Of-course, there are a number of granularities within these extremes, based on how many levels of loop nests we consider. For instance, suppose we have the schedule fragment $7(5A2(2B3C))$, where actor C has one output edge (connecting to an actor D not in this sub-schedule) on which it writes 1 token on each firing. At the finest level of granularity, the buffer on edge CD increases by one each time C fires. At the next level of granularity, we consider the 3 firings of C in the innermost loop together, and model the increase by steps of 3. Going up by one more level, the increase becomes steps of 6, and finally, at the coarsest level, we have an increase from 0 to 42 (=7*2*3*1) on the first firing of C.

In this paper, we assume the coarsest level of buffer modeling. The finer levels, although requiring less memory theoretically, may be practically infeasible to achieve due to increased complexity of the algorithms, and increased complexity of implementation in terms of maintaining pointers and memory allocations.

Initial tokens on edges can be handled very naturally in this model. An edge that has an initial token will have a buffer that is live right at the beginning of the schedule. It may be live for the entire dura-

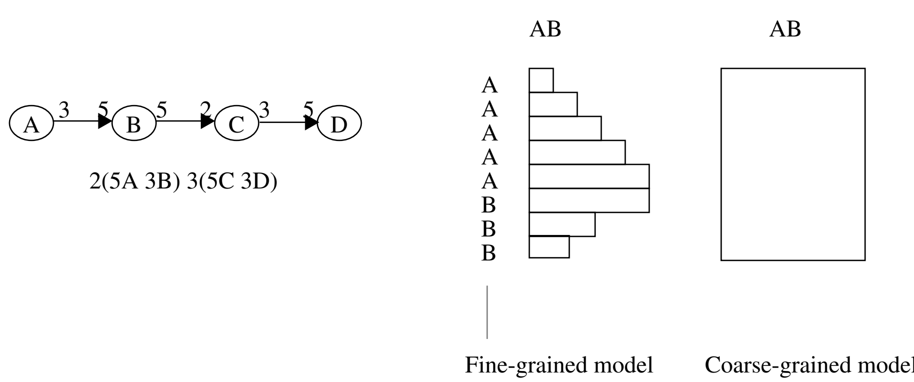


Fig 3. The fine-grained and coarse-grained models of buffer sharing.

tion of the schedule if the buffer never has zero tokens. If the buffer does have zero tokens at some point, then the buffer would be not be live for the portion of the schedule where the buffer has zero tokens.

Note that the best schedule under the non-shared buffering model is not necessarily the best schedule under the shared model as shown in figure 4.

As a first cut heuristic, the following dynamic programming formulation is immediately obvious:

$$bufmem(S_{1,n}) = MIN_{1 \le x \le n}\left\{MAX(bufmem(S_{1,x}), bufmem(S_{x+1,n})) + \sum_{e \in E_s} TNSE(e)\right\} \quad \textbf{(EQ 5)}$$

where the last term is again the sum of the buffer costs crossing the cut. The term $bufmem(S)$ is defined differently under the shared buffer memory model; in fact, equation 5 is the definition! Of course, $bufmem(S) = 0$ if S has only one actor.

The maximum of the left and right costs is taken based on the intuition shown in figure 5. Since the buffers in the subgraph on the right side of the cut will cannot be live at the same time as the buffers in the left side of the cut are live, and vice-versa, we only need the maximum of these two along with the buffers crossing the cut. The right buffers are overlayed with the left ones.

The formulation in equation 5 is not optimal because it makes a worst case assumption that all buffers crossing the cut are simultaneously live with all of the buffers on the left buffers and right buffers, thus preventing any sharing between them. For example, consider the example in figure 6. For the given schedule, the top level partition occurs on edge DE, with a cost of 36. According to the formulation, the total cost should be 36+MAX(b[A..D],b[E..F]). For b[A..D], the partition occurs on edge BC. Hence the

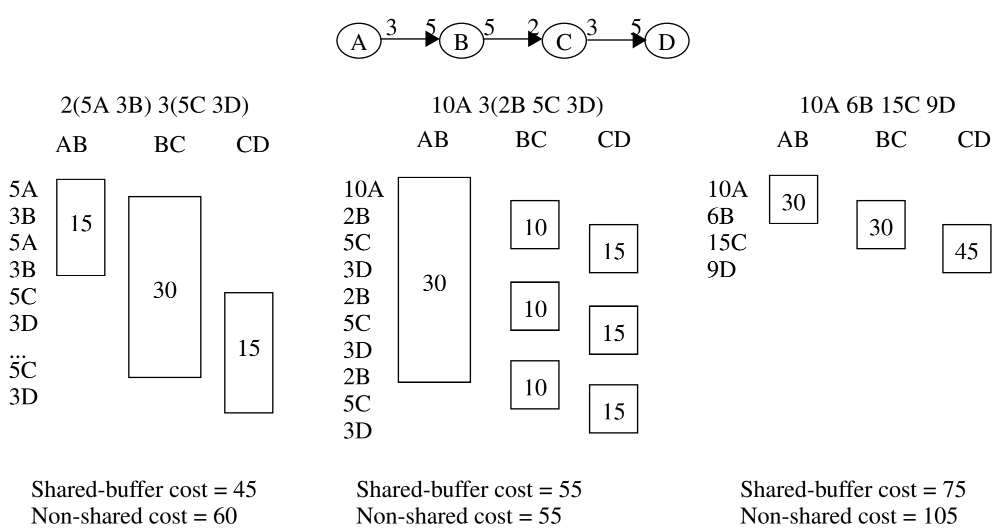


Fig 4. An example to show that the shared-buffer-optimal schedule is not the same as the non-shared-buffer optimal schedule.

cost is given by 84+MAX(20,7) = 104. For b[E..F], we have b[E..F] = 8. Hence the total cost gets computed as 36+MAX(104,8) = 140, while the actual cost is only 127.

5.1 Factoring loops

One of the central observations for proving the optimality of DPPO under the non-shared model is the following [19]:

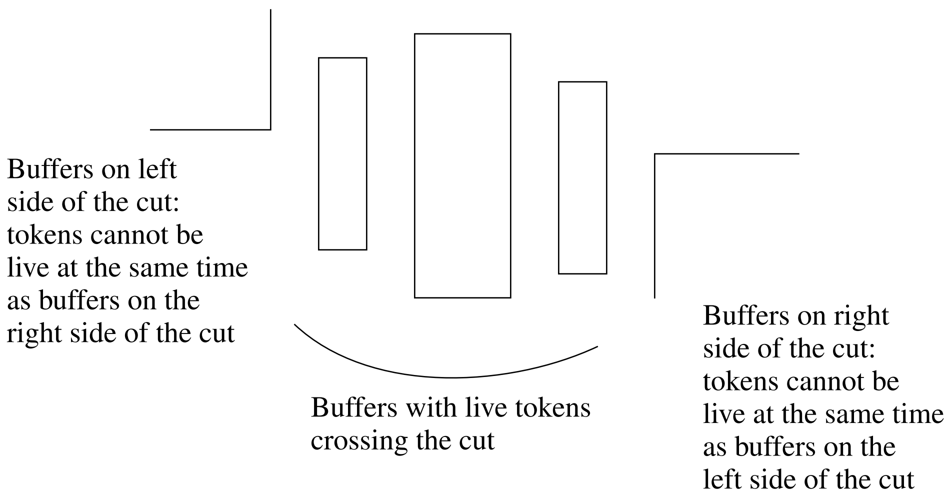


Fig 5. The intuition for a revised dynamic programming formulation.

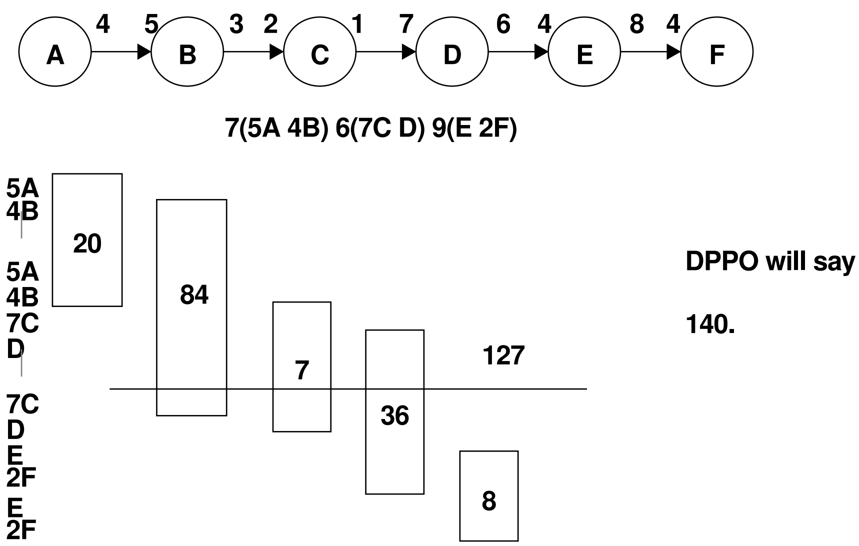


Fig 6. An example to illustrate the need for better accounting of costs.

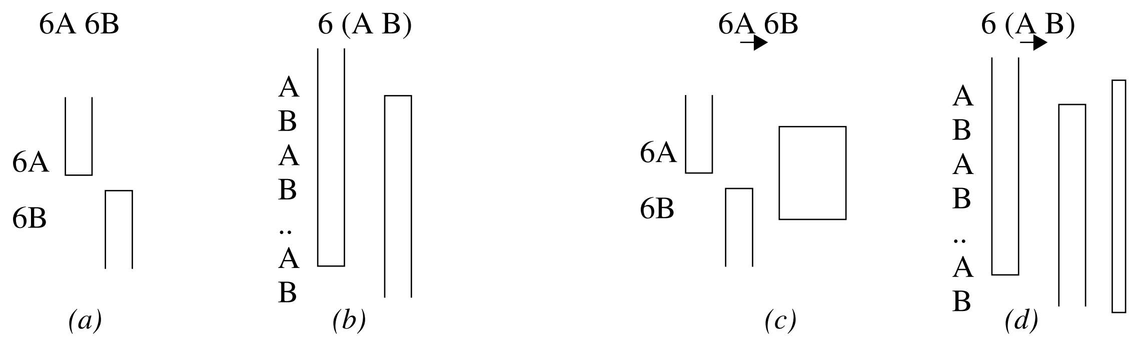


Fig 7. An example to illustrate that factoring can increase buffering requirement under the shared model.

Fact 1: Suppose that S is a valid schedule for an SDF graph G, and suppose that $L = (m(n_1S_1)(n_2S_2)\ldots(n_kS_k))$ is a schedule loop in S of any nesting depth such that $(1 \le i < j \le k) \Rightarrow actors(S_i) \cap actors(S_j) = \varnothing$. Suppose also that γ is any positive integer that divides $n_1, n_2, \ldots, n_k$; let L' denote the schedule loop $(\gamma m(\gamma^{-1}n_1S_1)(\gamma^{-1}n_2S_2)\ldots(\gamma^{-1}n_kS_k))$; and let S' denote the schedule that results from replacing L with L' in S. Then

(a). S' is a valid schedule for G; *and*

(b). $bufmem(S') \le bufmem(S)$. ($bufmem$ as defined by equation 1).

Unfortunately, this observation breaks down under the shared-buffer model, and we cannot automatically assume that factoring will not make things worse. Indeed, consider the example in figure 7. In figure 7 (a) and (b), there is no edge between actors A and B. If we do not factor the schedule, as shown in figure 7(a), we see that buffer profiles will be such that the buffers on the output edges of actor B will be disjoint from the buffer on the input edges of actor A. If we factor the schedule, as shown in figure 7(b), these buffers will no longer be disjoint, thus preventing sharing. Moreover, no advantage is gained from the factoring since the sizes of the input and output buffers do not decrease; factoring reduces the size of buffers only on edges between the actors being merged. On the other hand, if there is an edge (or more) between actors A and B, as shown in figure 7 (c) and (d), then factoring can reduce the overall buffering requirement if the reduction of the size of the buffer on edge (A, B) is more than the increase due to the overlap of the input and output buffers. Since this depends on the actual parameters of the graph, we follow a simple heuristic in the DPPO formulation above: we don't factor a loop if there are no internal edges (that is, edges whose terminal points are all actors that are being merged). We factor if there are internal edges, even though this might sometimes be suboptimal (if the reduction of the buffer sizes on the internal edges is less than the increase due to the overlap of the input and output buffers). Of-course, we could attempt to compute this increase or decrease, but this would in general require us to do a partial memory allocation using dynamic storage allocation techniques. This would greatly increase the complexity of the algorithms; we choose to use the simpler approach of using the heuristic approach to determine whether to factor or not.

6 A precise dynamic programming formulation for chain structured graphs

To improve the formulation given in equation 5, we need to keep better track of which buffers are actually live with the cut-crossing buffers. We first develop a technique for chain-structured graphs, and then extend this to arbitrary graphs.

Instead of having a single number as a cost, we define the cost to be a triple

$$(left, \cos t, right)$$

Let the chain-structured graph consist of actors $x_1, \ldots, x_N$. Consider a schedule for the subgraph consisting of actors $x_i, \ldots, x_j$. In the triple, $left$ is the size of the buffers in this subgraph that can be live simultaneously (or overlap) with the buffer on the input edge to actor x_i, the first actor in the subgraph. Similarly, $right$ is the size of the buffers that can overlap with the buffer on the output edge of actor x_j. For example, figure 6 makes this concept clearer.When we compute the cost for the subchain $ABCD$), we report the cost as $(104, 104, 91)$. This is because the buffer on edge DE can only overlap with the buffers BC and CD, as shown in the figure. The cost for implementing $ABCD$ in isolation is 104. Now when the cost for $ABCDEF$ is computed, the cost taken into consideration for the subchain $ABCD$ is 91, rather than 104, and this will lead to the correct value of 127 being computed. Hence, for the precise formulation, we will need to define exactly the rules for adding these triples.

6.1 Rules for adding triples

There are six cases to consider, based on the repetition counts of the left and right subgraphs in relation to the repetition count of the overall loop under consideration. Consider the subgraph $x_i, \ldots, x_j$. Define $g_{ij} = gcd(q_i, \ldots, q_j)$, where q_i is the repetition count of actor x_i in a complete period of the SDF schedule for the entire graph. The overall structure of the dynamic programming algorithm remains the same: we consider the cost of performing the partition at each k between i and $j-1$ inclusive. Because of the bottom up computation, we already know the costs $b[i, k]$ and $b[k+1, j]$ for each such k. We want to determine

$$(t_1, t_2, t_3) = (l_1, l_2, l_3) + c_{ij}(k) + (r_1, r_2, r_3)$$

Case 1

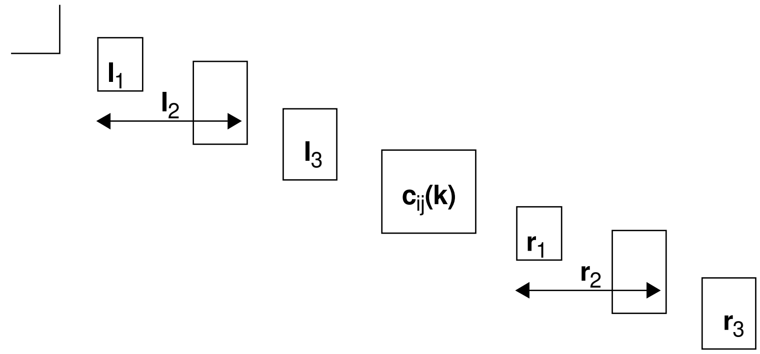


Fig 8. Buffer profiles for case 1.

where $c_{ij}(k)$ is the size of the buffer crossing the partition at k, for the subchain of actors $x_i, \ldots, x_j$.

6.1.1 Case 1: $g_{ik} = g_{ij}$ $g_{i(k+1)} = g_{ij}$

For this case, we have

$$t_1 = l_1 \qquad t_2 = max(l_2, l_3 + c_{ij}(k), r_1 + c_{ij}(k), r_2) \qquad t_3 = r_3$$

The reasoning for this made clearer by figure 8. Without loss of generality, we assume that l_2 reflects the cost by including l_1. Denote the schedule for the subchain consisting of actors $x_i, \ldots, x_j$ as S_{ij}. Because of the condition on the gcd, the schedule loop containing the actors $x_i, \ldots, x_k$ is iterated only once in the overall schedule loop containing the actors $x_i, \ldots, x_j$. In other words, $S_{ij} = S_{ik}S_{(k+1)j}$. The buffer crossing the split is from actor x_k to x_{k+1}; this buffer will overlap only with the right component of the cost of S_{ik}. This means that the left part of the cost, the cost presented to anything that might come on the left side of the schedule for actors $x_i, \ldots, x_j$, is going to remain the same as the left component of S_{ik}. Similarly, the right component of the cost of S_{ij} remains the same as the right component of $S_{(k+1)j}$. The central component of S_{ij}, as shown in figure 8 will be given by $max(l_2, l_3 + c_{ij}(k), r_1 + c_{ij}(k), r_2)$ because, as already mentioned, the split-crossing buffer only overlaps with the right component of S_{ik} and the left component of $S_{(k+1)j}$.

6.1.2 Case II: $g_{ik} = 2g_{ij}$ $g_{i(k+1)} = g_{ij}$

In this case, the schedule $S_{ij} = (2S_{ik})(S_{(k+1)j})$, and the buffer profile looks as shown in figure 9. The cost is now given by

$$t_1 = max(l_1 + c_{ij}(k), l_2) \qquad t_2 = max(l_2 + c_{ij}(k), r_1 + c_{ij}(k)) \qquad t_3 = r_3.$$

As we can see from figure 9, the buffer that will be on the left of the subchain consisting of $x_i, \ldots, x_j$ will be on the edge entering x_i, and since there are two invocations of S_{ik} in S_{ij}, this buffer will overlap with l_2 or $l_1 + c_{ij}(k)$.

6.1.3 Case III: $g_{ik} > 2g_{ij}$ $\quad g_{i(k+1)} = g_{ij}$

The buffer profile is as shown in figure 10. The costs are given by

$$t_1 = l_2 + c_{ij}(k) \qquad t_2 = max(l_2 + c_{ij}(k), r_1 + c_{ij}(k)) \qquad t_3 = r_3$$

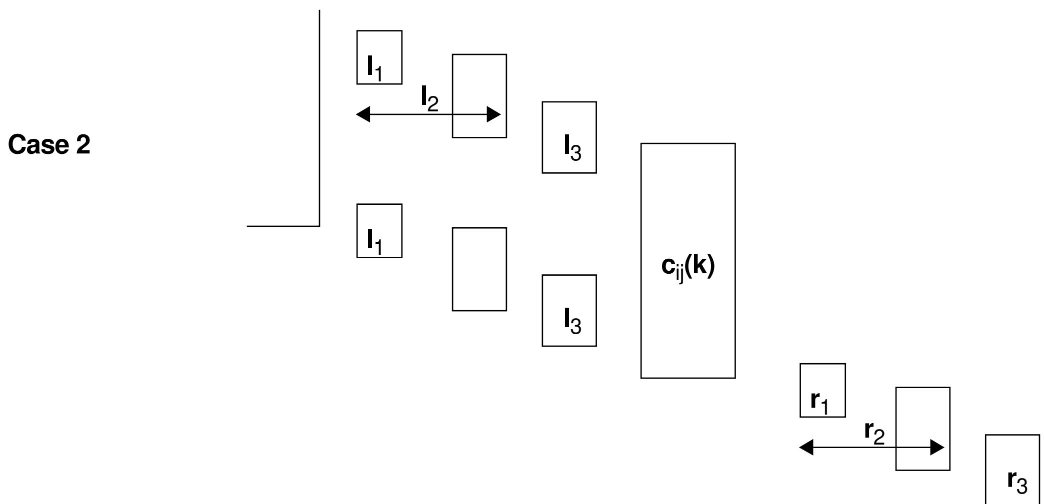


Fig 9. Buffer profiles for case 2.

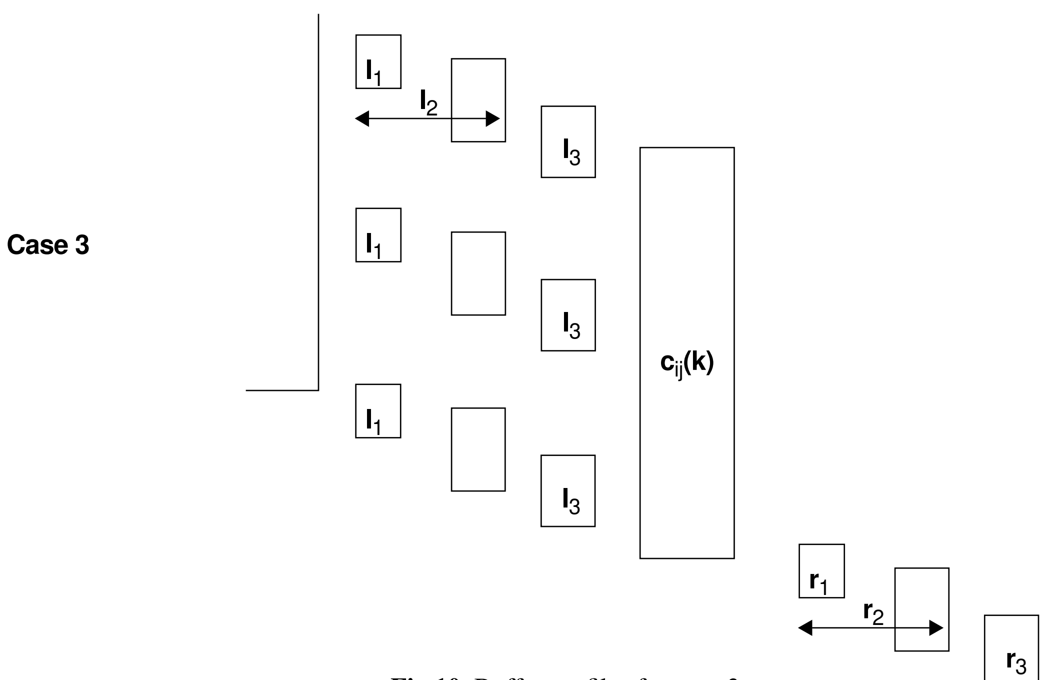


Fig 10. Buffer profiles for case 3.

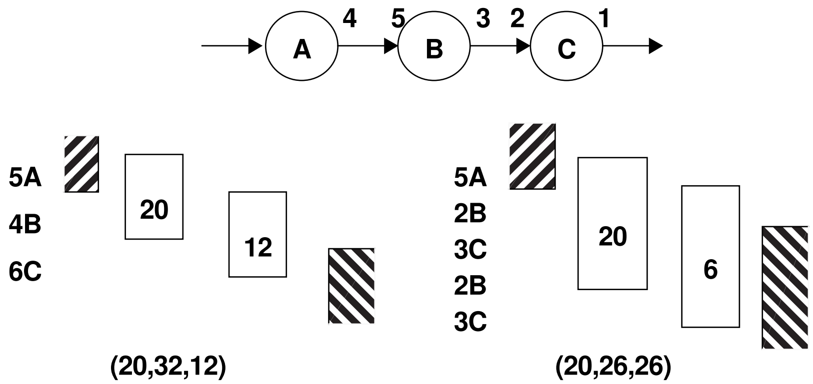


Fig 11. Two schedules for the sub-chain shown. The costs are incomparable.

The only difference from the previous case is that now the left component is unambiguously $l_2 + c_{ij}(k)$ since during the second invocation of S_{ik}, the overlap of l_2 with $c_{ij}(k)$ occurs as shown in figure 10.

There are nine cases in all to be considered, since each of g_{ik} and $g_{(k+1)j}$ has three possibilities that lead to distinct costs. The other six cases can be analyzed in a similar way as for the three cases dealt with above, and we omit their description.

The MIN operation on these tuples is defined simply by comparing each element in the tuple. Hence, we can have incomparable tuples where some elements are smaller while some are larger. In such cases, we will have to "carry" along all incomparable tuples until they can be eliminated. An example of how incomparable tuples arise is shown in figure 11. The first schedule, which has the smaller right cost, will be the preferred schedule provided that the schedule $5A\ 4B\ 6C$ is never nested in any schedule loop of loop count more than 1. Otherwise, the other schedule will result in a lower cost. We follow the strategy of simply recording both in the dynamic programming table, and resolving the issue when subgraphs of four or more actors are considered. It is technically possible that there may be a multiplicative growth in the number of incomparable tuples but this phenomenon has not been observed in practice. Of-course, we can always impose a bound on the number of incomparable tuples that are retained, in order to keep space and running time polynomially bounded.

Unfortunately, extending this procedure for arbitrary acyclic graphs has not proven to be possible. Hence, for arbitrary graphs, we rely on the heuristic DPPO formulation described in section 5 to compute the best loop nesting hierarchy for a given SAS.

7 Generating single appearance schedules

In [3], it has been shown that for an arbitrary acyclic graph, a SAS could be derived from a topological sort of the graph. To be precise, the class of SASs for a delayless acyclic graph can be generated by enumerating the topological sorts of the graph. We use the lexical ordering given by each topological sort to derive a flat SAS (this is a schedule of the form $(q_1x_1)(q_2x_2)\ldots(q_nx_n)$, where the x_i are actors and q_i are there repetitions. The lexical order $x_1x_2\ldots x_n$ is the order given by the topological sort of the graph.). This schedule then leads to a set of nesting hierarchies; the complete set of lexical orders and for each lexical order, the set of nesting hierarchies, constitutes the entire set of SASs for the graph. Hence, we need a method for generating the topological sort. As shown in [3], the general problem of constructing buffer-optimal SASs under both models of buffering, namely the coarse shared buffer model, and the non-shared model, is NP-complete. Thus, the methods for generating topological sorts are necessarily heuristic, and not-optimal in general.

In [3], two methods have been developed: a bottom-up method based on clustering called APGAN, and a top down method based on graph partitioning called RPMC. The heuristic rule-of-thumb used in RPMC is to find a cut of the graph such that all edges cross in the same direction (enabling us to recursively schedule each half without introducing deadlock), and such that size of the buffers crossing the cut is minimized. While this rule is intuitively attractive for the non-shared buffer model, it is also attractive for the shared model since the cut-crossing buffers will not be disjoint, and cannot be shared. Hence, it makes intuitive sense to minimize the size of these buffers, and RPMC is a reasonable heuristic to use even under the shared buffer model.

The APGAN technique is based on clustering adjacent nodes together that communicate heavily, so that these nodes will end up in the innermost loops of the loop hierarchy. For a broad subclass of SDF systems, APGAN has been shown to construct SAS that provably minimize the non-shared buffer memory metric over all SAS [3].

8 Creating the interval instances from a single appearance schedule

Given a SAS, we have to solve the problem of deriving the set of intervals corresponding to the buffer lifetimes. These lifetimes could also be periodic; it would be desirable to represent the periodicity implicitly, without having to physically create an interval for each occurrence. For allocating periodic intervals, we assume that all instances of a particular lifetime are allocated at the same block of memory. Hence, it suffices to allocate the first instance.

8.1 R-Schedules and the Schedule Tree

As shown in [19], it is always possible to represent any SAS for an acyclic graph as

$$(i_L S_L)(i_R S_R) \qquad \textbf{(EQ 6)}$$

where S_L and S_R are SASs for the subgraph consisting of the actors in S_L and in S_R, and i_L and i_R are loop factors for iterating these schedules. In other words, the graph can be partitioned into a left subset and a right subset so that the schedule for the graph can be represented as in equation 6. SASs having this form are called R-schedules [19].

Given an R-schedule, we can represent it naturally as a binary tree; we call this the **schedule tree**. The internal nodes of this tree will contain the loop-factor of the subschedule rooted at that node. The leaf nodes will contain the actors, along with their residual loop-factor. Figure 12 shows schedule trees for the SAS in figure 2. Note that a schedule tree is not unique since if there are loop factors of 1, then the split into left and right subgraphs can be made at multiple places. In figure 12, the schedule tree for the flat SAS in figure 2(b)(3) is based on the split $\{A\}\{B, C\}$. However, we could also take the split to be $\{A, B\}\{C\}$. However, the split will not affect any of the computations we perform using the tree.

In order to compute things like the “start time” and “end time” of a buffer lifetime, we use the following abstract notion of time: each invocation of a leaf node in the schedule tree is considered to be one schedule step, and corresponds to one unit of time. For example, the looped schedule $2(A\ 3B)$ would be considered to take 4 time steps. The first invocation of A would take place at time 0, and the last invocation of $3B$ begins at time 3 and ends at time 4.

8.2 Computing the duration times of buffer lifetimes

If v is a node of the schedule tree, then $subtree(v)$ is the (sub)tree rooted at node v. If T is a subtree, define $root(T)$ to be the root node of T. The function $loop: V \to Z$, where V is the set of nodes in the tree, and Z is the set of positive integers, returns for a non-leaf node, the loop iterator value at that nesting level, and returns 1 for a leaf node. The function $loop$ is used to compute the duration times, $dur(v)$ for all nodes v (i.e, loop nests) by depth-first-search on the tree:

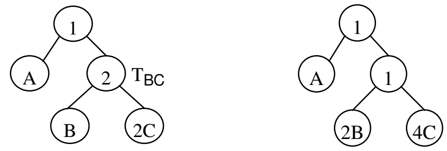


Fig 12. Schedule trees for schedules in figure 2(b)(2) and (3).

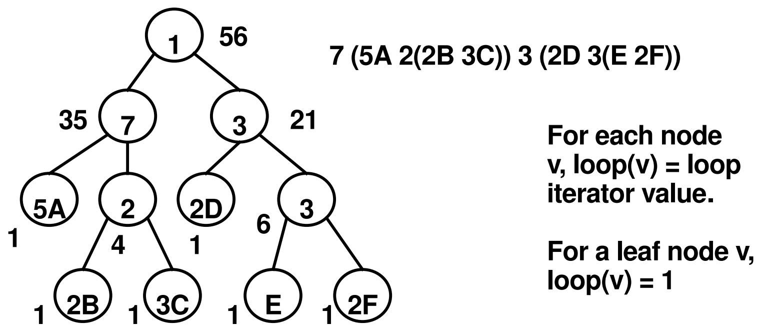


Fig 13. The binary tree representation of a SAS, along with the duration values for the nodes.

$$dur(v) = loop(v)(dur(left(v)) + dur(right(v)))$$

where $right(v)$ $(left(v))$ is the right (left) child of node v. For leaf nodes, $dur(v) = 1$. The numbers beside the nodes in figure 13 show the result of the depth-first-search computation of the duration values. The depth first search takes time $O(|V|)$.

8.3 Computing the start and end times of buffer lifetimes

The next task is to compute the start and stop times for each nested loop. This is also done using depth-first-search (also taking time $O(|V|)$), as shown by the pseudocode in figure 14. An example of a SAS with the computed start and stop times is shown in figure 15.

The final task is to compute the lifetimes for the buffers. We introduce some notation first:

```
Procedure computeStartStopTimes(Tree* schedule) {
      // Initialize
      visited(v) ← 0        ∀v
      doComp(root,0)
}
Procedure doComp(v, start) {
      start(v) ← start;stop(v) ← start + dur(v)
      if (¬visited(left(v)) doComp(left(v), start);
      if (¬visited(right(v)) doComp(right(v), stop(left(v)))
      visited(v) ← 1
}
```

Fig 14. Pseudocode for the depth-first-search algorithm for computing start and stop times for all the nested loops.

Definition 1: A **parent** of a pair of nodes v_1, v_2 is any node that contains the nodes v_1, v_2 as leaf nodes rooted at that node.

Definition 2: The **smallest parent** of a pair of nodes v_1, v_2 is the *first* node that contains nodes v_1, v_2 as leaf nodes rooted at that node, where first is measured from the leaf nodes.

Definition 3: The **greatest parent** of a pair of nodes v_1, v_2 is the last node that contains nodes v_1, v_2 as leaf nodes rooted at that node such that all parents of this node have a value of unity.

Definition 4: The **parent set** of a pair of nodes v_1, v_2 is the set of all parents of v_1, v_2.

Fact 2: Every node in the parent set lies on the path from the smallest parent to the largest parent.

The smallest and greatest parents of a pair of leaf nodes correspond to the innermost and outermost loops that contain the actors corresponding to the leaf nodes. The start time of the lifetime of a buffer on an edge (u, v) is clearly the start time computed for the leaf node in the schedule tree, corresponding to actor u. The stop time of the buffer interval is first time is stops being live. Note that an interval can be periodic and become live again later on. We are interested in the first time it stops being live since that quantity, along with the periodicity parameters will completely characterize the interval. This stop time, however, is not simply the stop time of the leaf node corresponding to the sink actor. The algorithm shown in figure 16 is used to compute the stop time, and it takes $O(|V||E|)$ time.

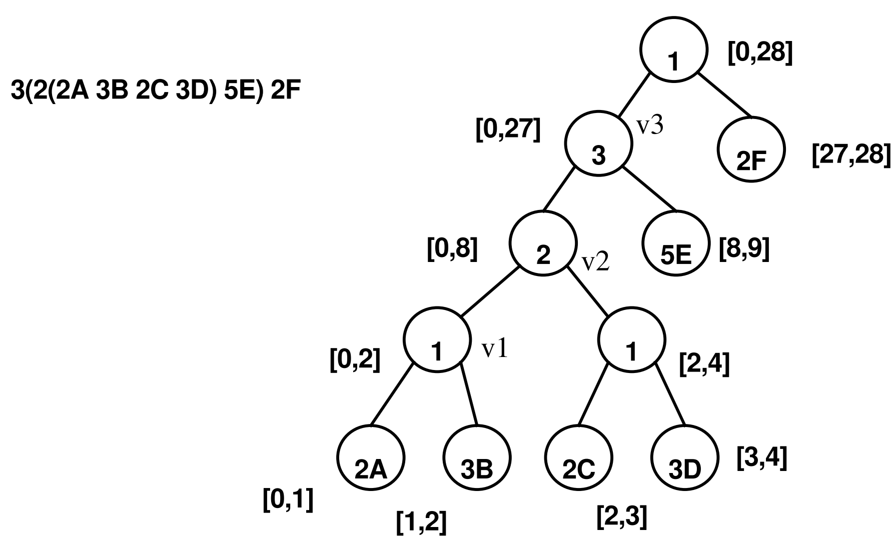


Fig 15. Start and stop times for each node computed using depth-first-search.

```
Procedure computeIntervalStopTime

foreach buffer(u, v)
      //find the least parent of (u,v) in scheduleTree
      leastParent ← findLeastParent(u, v)
      stop_uv ← stop(right(leastParent))
      tmp ← vleaf // vleaf = leaf node corresponding to actor v
      while (tmp ≠ right(leastParent))
            if (left(parent(tmp)) = tmp)
                  stop_uv ← stop_uv – dur(right(parent(tmp)))
            fi
            tmp ← parent(tmp)
      end while
      // stop(buffer(u,v)) sets the stop time of buffer(u,v)
      stop(buffer(u, v)) ← stop_uv
end for
```

Fig 16. Procedure for computing the earliest stop time of an interval.

8.4 Computing the periodicity parameters of buffer lifetimes

Given a schedule, the buffer on each edge in the SDF graph has a particular lifetime profile. This profile can be periodic as shown in figure 17. The periodicity arises due to the nested loops. By periodic, we mean that the lifetime is fragmented in a deterministic, predictable manner. More precisely, the times during which the buffer is live can be described much more succinctly than by simply enumerating all the occurrences of the live portions. It is useful to keep track of this periodicity in certain cases since two buffers could have disjoint lifetimes that can be shared, as shown in figure 17 for buffers on edges (A, B) and (C, D). If the buffer on an edge e is not disjoint with any other buffer, then any periodicity it has can be ignored, and it can be considered to be live the entire time over all of its periods.

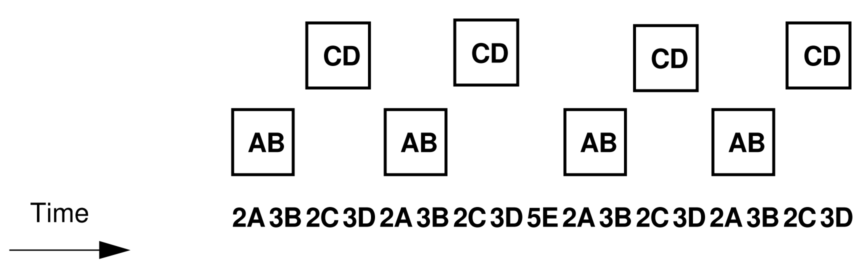


Fig 17. A couple of periodic intervals corresponding to the schedule from figure 15.

For a buffer b on an edge (U, V) in the SDF graph, let the parent set be denoted as nodes $v_1^b, \ldots, v_n^b$, with v_1^b is the smallest parent. For example, in figure 15, say there is an edge between actors A and B. The smallest and greatest parents are the nodes marked $v1$ and $v3$. The node $v2$ is also in the parent set, and is on the path from $v1$ to $v3$. We represent a periodic lifetime of a buffer b by a triple consisting of two tuples and an integer of the form

$$\{start(v_1^b), (a_1^b, \ldots, a_n^b), (loop(v_1^b), \ldots, loop(v_n^b))\}$$

where a_i^b are the constants $dur(left(v_i^b)) + dur(right(v_i^b))$. Clearly, nodes in the parent set that have $loop$ values of unity will not contribute to the periodicity; hence, those components can be eliminated. This triple allows us to represent the buffer lifetime in the following way: buffer b is live for time intervals

$$[start(v_1^b) + p_1^b a_1^b + \ldots + p_n^b a_n^b \quad , \quad start(v_1^b) + p_1^b a_1^b + \ldots + p_n^b a_n^b + dur(b)]$$

for all combinations of $p_i^b \in \{0, \ldots, loop(v_i^b) - 1\}$. For example, for the buffer AB in figure 17, we have by inspection that $start(\ldots) = 0$, $dur(AB) = 2$, $(a_1^b, \ldots, a_n^b) = (4, 9)$, and $\{loop(\ldots)\} = \{2, 2\}$. Hence, buffer AB is live during the intervals given by $[4x + 9y, 4x + 9y + 2]$ for all combinations of $x \in \{0, 1\}$ and $y \in \{0, 1\}$. The live intervals are $[0, 2]$, $[4, 6]$, $[9, 11]$, and $[13, 15]$.

During storage allocation, we will have to determine whether, at a time T, a buffer b is live or not. In essence, we have to determine whether the equation

$$start(v_1^b) + p_1^b a_1^b + \ldots + p_n^b a_n^b \le T \le start(v_1^b) + p_1^b a_1^b + \ldots + p_n^b a_n^b + dur(b) \qquad \textbf{(EQ 7)}$$

where the p_i^b are variables that range over $\{0, \ldots, loop(v_i^b) - 1\}$ has a solution in p_i^b. A solution, if it exists, can be found easily because of the following obvious property on the p_i^b. Let the a_i^b be sorted in increasing order. Then, the following has to hold:

$$a_i^b(loop(v_i^b) - 1) \le a_{i+1}^b \qquad i = 1, \ldots, n - 1 .$$

Given this observation, equation 7 can be solved by the algorithm in figure 18.

If b is not alive at time T, we will need to determine when the next instance of its periodic interval will occur again. This computation is needed to determine whether some other interval of a particular duration is completely disjoint with the set of intervals corresponding to b. That is, determine whether some other interval can be fitted into the same location that b might be assigned to. The starting time of the next instance of the periodic interval is obtained simply by incrementing the “number” formed by the k_i in the basis $(loop(v_1^b), \ldots, loop(v_n^b))$. For example, let $(loop(v_1^b), \ldots, loop(v_n^b))$ be $(2, 2, 2)$, let $(a_1^b, \ldots, a_n^b)$ be $(28, 13, 4)$, and let (k_i) be $(0, 1, 1)$. The number this represents is clearly

$0 \cdot 28 + 1 \cdot 13 + 1 \cdot 4 = 17$. The next time this buffer will be live again will be given by incrementing this "number" by one, in the basis $(2, 2, 2)$: $(0, 1, 1) + 1 = (1, 0, 0)$. This gives 28 as the next starting time.

Given this method for testing whether a periodic buffer is live at a given time, we can easily test whether two periodic buffers are disjoint, or whether they intersect. The test would take time $O(|V|)$ in the worst case, where V is the set of actors in the SDF graph. The reason is that an SAS that has a schedule tree of linear depth (i.e, a depth of $|V| - 1$) would have a parent set of $O(|V|)$ nodes for any buffer between actors in the innermost loop. Hence, in the procedure in figure 18, $n = O(|V|)$, and the test takes time $O(|V|)$. However, on average, it is more likely that the schedule tree will have logarithmic depth; in such cases, the running time of the testing procedure will be $O(\log|V|)$. The next step is to allocate the various buffers to memory.

9 Dynamic Storage Allocation

Once we have all of the lifetimes, we have to do the actual assignment to memory locations of the buffers. This assignment problem is called dynamic storage allocation (DSA) and is formally defined as:

Definition 5: Let B be the set of buffers. Let $N = |B|$, the number of elements in B. For each $b \in B$, $s(b)$ is the time at which it becomes live, $e(b)$ is the time at which it dies, and $w(b)$ is the **size** of buffer b. Note that the **duration** of a buffer is $e(b) - s(b)$. Given the s, e, w values for each $b \in B$, and an integer K, is there an allocation of these buffers that requires total storage of K units or less? By an allocation, we mean a function $A{:}B \to \{0, \ldots, K-1\}$ such that $0 \le A(b) \le K - w(b)$ for each $b \in B$, and if two intervals b_1 and b_2 intersect (using the intersection test for periodic buffer lifetimes as described earlier) then $A(b_1) + w(b_1) \le A(b_2)$ or $A(b_2) + w(b_2) \le A(b_1)$.

Define $T' \leftarrow T - start(v_1^b)$.

for i = n to 1 step -1

$$k_i \leftarrow min\left(\left\lfloor \frac{T'}{a_i^b} \right\rfloor, loop(v_i^b)\right)$$

$$T' \leftarrow T' - k_i a_i^b$$

end for

if ($T' < dur(b)$), then b is live (the computed k_i are the solution)

else not live.

Fig 18. An algorithm to determine whether a periodic buffer is live at a particular time.

The “dynamic” in DSA refers to the fact that many times, the problem is online in nature: the allocation has to be performed as the intervals come and go. For SDF scheduling, the problem is not really “dynamic” since the lifetimes and size of all the arrays that need to be allocated are known at compile time; thus, the problem should perhaps be called static storage allocation. But we will use the term DSA since this is consistent with the literature.

Theorem 1: [9] DSA is NP-complete, even if all the sizes are 1 and 2.

9.1 Some notation

An **instance** is a set of buffers. An **enumerated instance** is an instance with some ordering of the buffers. For an instance, we have associated with it a **weighted intersection graph** (**WIG**) $G_B = (V_B, E_B)$ where V_B is the set of buffers, and E_B is the set of edges. There is an edge between two buffers iff their lifetimes overlap in time. The graph is node-weighted by the sizes of the buffers. For any subset of nodes $U \subset V_B$, we define the weight of U, $w(U)$ to be the sum of the sizes $w(v)$ for all $u \in U$. A **clique** is a subset of nodes such that there is an edge between every pair of nodes. The **clique weight** (**CW**) is the weight of the clique. The **maximum clique weight** (**MCW**) in the WIG is the clique with the largest weight, and is denoted $\tilde{\omega}(G_B)$. The MCW corresponds to the maximum number of values that are live at any point. The **chromatic number (CN)**, denoted $\chi(G_B)$, for G_B is the minimum K such that there is a feasible allocation in definition 5.

First fit (FF) is the algorithm that performs allocation for an enumerated instance by assigning the smallest feasible location to each interval in order. It does not reallocate intervals, and it does not consider intervals not yet assigned. The pseudocode for this algorithm is shown in figure 19. In order to analyze the running time, we observe that $N = |E|$, and $|E| = O(|V|)$ for sparse SDF graphs, where V, E are the node and edge sets for the SDF graph. Hence, building the weighted intersection graph takes $O(|V|^3)$ time in the worst case (all buffers overlap with each other, schedule tree is of linear depth), and time $O(|V|^2 \cdot \log|V|)$ if the schedule tree is of logarithmic depth. The *foreach* loop of the `firstFit` procedure takes time $O(|V|^2 \cdot \log(|V|))$ in the worst case if every buffer overlaps with every other buffer; hence, the `firstFit` procedure has running time dominated by the `buildIntersectionGraph` step.

We refer the reader to [20] and references therein for a more detailed treatment of this very interesting DSA problem. The study in [20] shows that in practice, firstfit is a good heuristic, and we use it in our compiler framework here. In particular, the empirical study on random WIGs in [20] shows that ordering the buffers by durations gives the best results, followed by ordering using start times. Hence, in our

```
Procedure FirstFit(enumerated instance I)

G = buildIntersectionGraph(I)
Vector allocate //allocate is an array to contain the allocations

foreach interval i in I do
      allocate(i) ← 0  //initial allocation at 0
      neighborsAllocations ← { }
      foreach neighbor j of i from G
            if (j appears before i in I)
                  neighborsAllocations ← neighborsAllocations ∪ {allocate(j)}
            fi
      end for
      sort(neighborsAllocations)
      foreach allocation a ∈ neighborsAllocations
            if allocate(i) conflicts with a
                  //size(a) = size of interval with allocation a
                  allocate(i) ← a + size(a)
            fi
      end for
end for



Procedure buildIntersectionGraph(enumerated instance I)

sort I by start times
N ← number of buffer lifetimes in I
// G is an adjacency list representation containing N rows
// and list pointer at each G(i)
Graph G

foreach i in {1,...,N}
      j ← i + 1
      while (start time of I(j) < end time of I(i))
            if (lifetime(I(j)) overlaps lifetime(I(i)))
                  G(i) ← G(i) ∪ {j}
                  G(j) ← G(j) ∪ {i}
            fi
            j ← j + 1
      end while
end for
```

Fig 19. Pseudocode definition of the FirstFit heuristic.

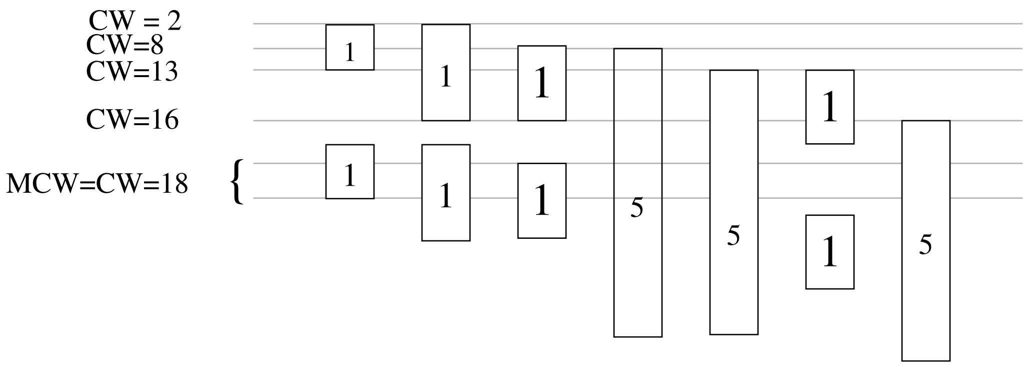


Fig 20. An example that shows that the MCW can occur at a time that is not the earliest start time of any interval.

experiments in section 10, we will apply firstfit on both ordering by start times (abbreviated *ffstart*), and ordering by durations (*ffdur*).

It is clear that the MCW is a lower bound on the CN of an WIG. It is known that the CN can be as much as 1.25 times the MCW for particular instances; however, it is not known whether 1.25 is a tight upper bound. The MCW is thus a good lower bound to compare the performance of an allocation strategy on a particular set of lifetimes. Given that the experiments on random instances in [20] show that *ffdur* comes within 7% on average of the MCW, in practice, the CN is not much bigger than the MCW, certainly not as much as 1.25 times as big. Hence, we use the MCW for comparison purposes in our experiments in the next section.

While the MCW can be computed easily and exactly for an instance without periodic lifetimes, computing it for instances with periodic buffer lifetimes is more difficult. Consider the case where all intervals are non-periodic. Let MT be the set of all times (i.e, schedule steps) where there is maximum overlap of the intervals; that is, where the overlap amount is equal to the maximum clique weight. It is easy to see that MT must contain the start time of some interval. Hence, the maximum clique weight can be computed easily by sorting the intervals by their starting times, and determining the overlap at each starting time.

Now suppose that some of the intervals are periodic. It is still the case that MT will contain the start time of at least one interval; however, this need not be the earliest start time. It could be the start time of some periodic occurrence (greater than the earliest start time) of the interval (see figure 20). Hence, to compute the maximum clique weight in this scenario, we would have consider start times of all occurrences of a periodic interval; this becomes a non-polynomial time algorithm and could potentially take a long time if there are many periodic occurrences. Hence, in our experiments, we use two heuristics to compute these values. The first heuristic gives an optimistic estimate; it only considers the earliest start time of each interval, and it determines whether there is any overlap with other intervals at that time by using the

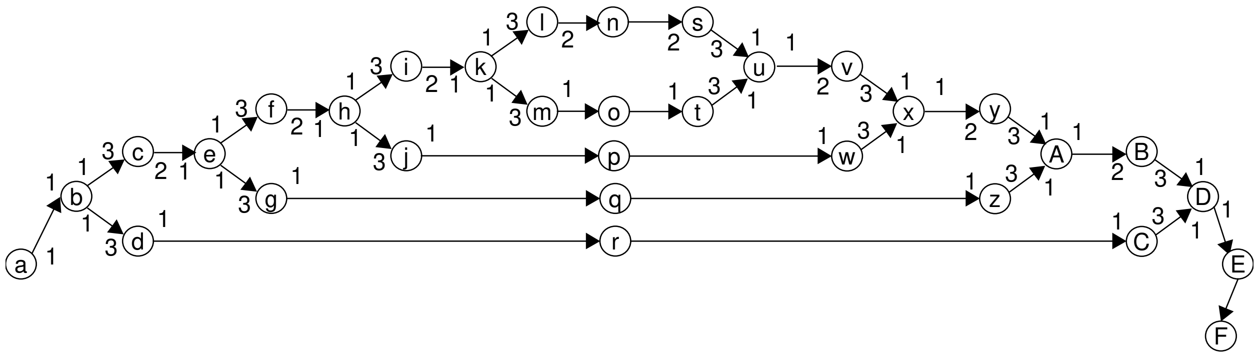


Fig 22. SDF graph for a one-sided filterbank of depth 4. The produced/consumed numbers not specified are all unity.

algorithm of figure 18. This is an optimistic estimate since the maximum clique weight could occur at a time that is not the earliest start time of any interval. The second heuristic gives a pessimistic estimate; it simply ignores the periodicity of periodic intervals, and assumes that a periodic interval is live the entire time between its earliest start time, and the last stop time (that is, the stop time of the last occurrence of the interval).

10 Experimental Results

Figure 21 shows a flow chart representing the sequence in which these algorithms are applied We have tested these algorithms on several practical benchmark examples, as well as on random graphs.

10.1 Practical systems

The practical examples are a number of one-sided filterbank structures [26] as shown in figure 22, two-sided filterbanks [26], as shown in figure 23, and a satellite receiver example [24] as shown in figure 24. Another variation that occurs in practical systems are the rate change ratios; the figure 22 shows a filterbank with 1/3, 2/3 rate changes; these are the changes that occur across actors c and d for instance.

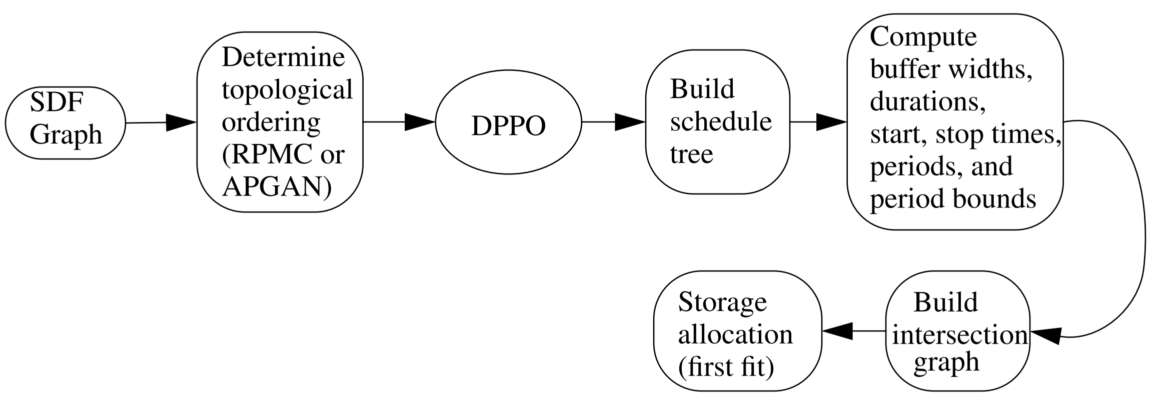


Fig 21. Flow chart showing the sequence in which the various algorithms are applied.

Other ratios that could be used include 1/2, 1/2, or 2/5, 3/5. Similarly, figure 23 shows a filterbank with 1/2, 1/2 rate changes, for example, across actors c and d. Again, these changes could be 2/3, 1/3, or 2/5, 3/5 for instance. Experimental data is summarized for these various parameters, as well as for filterbanks of different depths, in table 1. The leftmost column contains the name of the example. The filterbank "qmf23_2d", for example, is a filterbank of depth 2, with 1/3, 2/3 rate changes. Similarly, "qmf235_5d" is a filterbank of depth 5 with rate changes of 3/5 and 2/5. The depth 5, 3, and 2 filterbanks have 188, 44, and 20 nodes respectively. "nqmf23_4d" is the one-sided filterbank from figure 22. "satrec" is satellite receiver example from [24]. The other examples included are "16qamModem", an implementation of 16-QAM modem, "4pamxmitrec", a transmitter-receiver pair for a 4-PAM signal, "blockVox", an implementation of

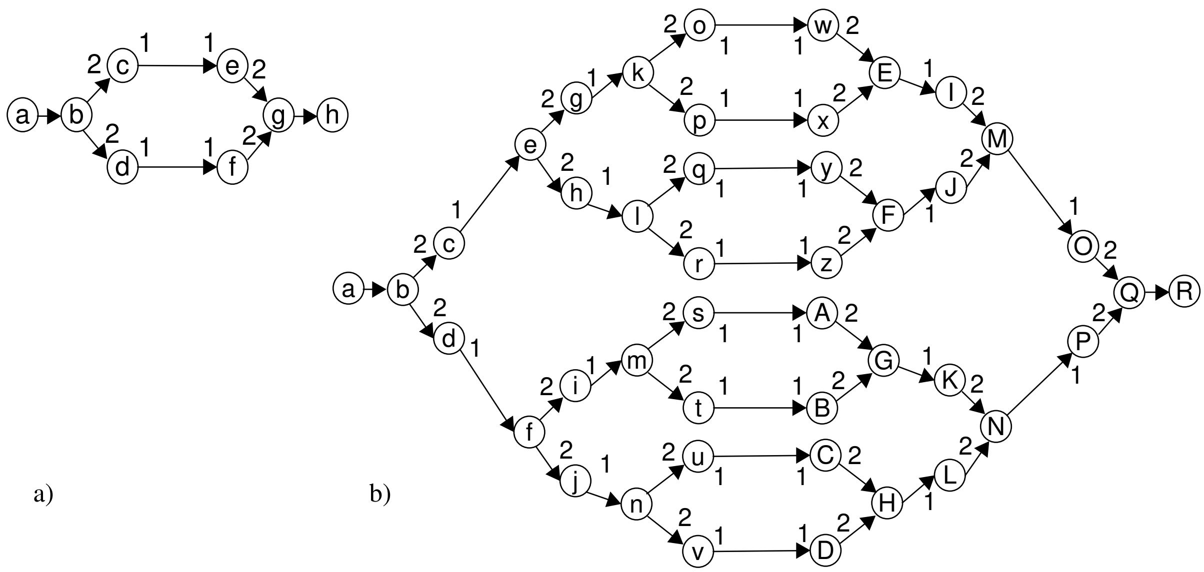


Fig 23. SDF graph for a two-sided filterbank. a) Depth 1 filterbank, b) Depth 3 filterbank. The produced/consumed numbers not specified are all unity.

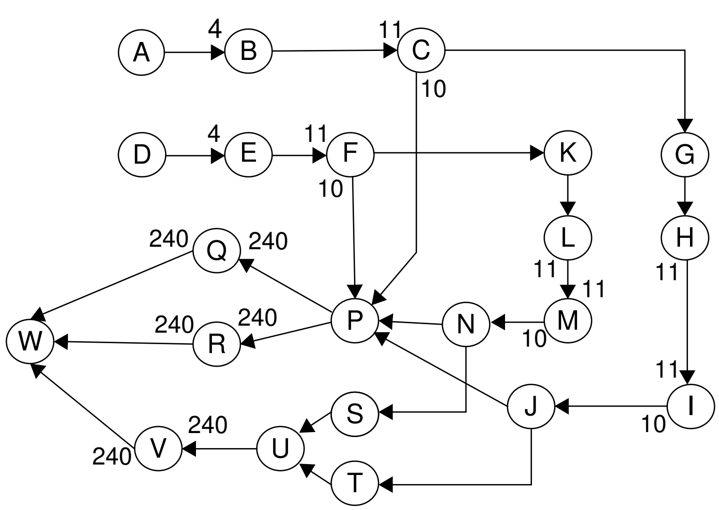


Fig 24. SDF abstraction for satellite receiver application from [24].

a vocoder (a system that modulates a synthesized music signal with vocal parameters—for instance, Stevie Wonder's "Part-time lover" has a sequence at the end of the song using this type of effect), "overAddFFT", an implementation of an overlap-add FFT (where the FFT is applied on successive blocks of samples overlapped with each other), and "phasedArray", an implementation of a phased array system for detecting signals. These examples are all taken from the Ptolemy system demonstrations [1].

The second column has the results of running RPMC and post-optimizing with DPPO on these systems, assuming the non-shared model of buffering. This column gives us the basis for determining the improvement with the shared model. In general, "(R)" refers to RPMC and "(A)" refers to APGAN. The third column has the results of applying the new heuristic DPPO (called sdppo) post-optimization for shared buffers on an RPMC generated schedule. The fourth and fifth columns contain optimistic (mco) and pessimistic (mcp) estimates of the maximum clique weight for the schedule generated by sdppo (on the RPMC generated schedule). The sixth and seventh columns contain the actual allocations achieved after applying the firstfit ordered by durations, and firstfit ordered by start times heuristics. The eigth column contains the BMLB [3] values for each system. Briefly, the buffer memory lower bound (BMLB) is a lower bound on the total buffering memory required over all valid SASs, assuming the non-shared model of buffering. The rest of the columns contain the results after applying these heuristics on APGAN-generated schedules. One each row, two numbers are shown in bold: the better DPPO result (RPMC or APGAN) and the best shared implementation (between *ffdur*(R), *ffstart*(R), *ffdur*(A), *ffstart*(A)). The last column has the percentage improvement over the non-shared implementation; this is computed as

$$\frac{MIN(dppo(R), dppo(A)) - MIN(ffdur(R), ffstart(R), ffdur(A), ffstart(A))}{MIN(dppo(R), dppo(A))} \cdot 100$$

As can be seen, the improvements average more than 50%, and are dramatic in some cases, with up to 83% improvement in the depth 5 filterbank of 1/2, 1/2 rate changes (the most common type of filterbank). It is interesting to note that in [24], the methods of Ritz et. al. for shared-buffer scheduling achieve an allocation of more than 2000 units for "satrec"; in contrast, the methods in this paper achieve 991, an improvement of more than 50%.

Table 1: Overall performance on practical examples

	dppo (R)	sdppo (R)	mco (R)	mcp (R)	ffdur (R)	ffstart (R)	bmlb	dppo (A)	sdppo (A)	mco (A)	mcp (A)	ffdur (A)	ffstart (A)	% impr.
nqmf23_4d	**209**	132	120	139	**132**	133	75	314	242	237	258	264	240	36.8
qmf23_2d	**60**	24	21	30	**22**	22	50	62	35	26	28	27	27	63.3

Table 1: Overall performance on practical examples

	dppo (R)	sdppo (R)	mco (R)	mcp (R)	ffdur (R)	ffstart (R)	bmlb	dppo (A)	sdppo (A)	mco (A)	mcp (A)	ffdur (A)	ffstart (A)	% impr.
qmf23_3d	**173**	63	54	90	**63**	64	116	188	104	74	90	78	78	63.6
qmf23_5d	**1271**	498	489	645	**492**	502	512	1478	812	741	902	792	804	61.3
qmf12_2d	36	12	9	9	**9**	9	34	**34**	16	11	12	12	11	73.5
qmf12_3d	88	21	16	19	**16**	17	78	**78**	35	25	36	27	27	79.5
qmf12_5d	434	72	56	72	**58**	63	342	**342**	142	103	165	113	113	83.0
qmf235_2d	**122**	55	50	65	**55**	66	82	140	85	71	75	74	79	54.9
qmf235_3d	**492**	240	220	285	**240**	248	192	660	431	380	392	382	394	51.2
qmf235_5d	**8967**	5690	5560	6065	**5690**	6226	852	13716	9248	7477	7635	8125	8254	36.5
satrec	2480	1920	1680	1680	1691	1715	1542	**1542**	1200	960	960	**991**	1015	35.7
16qam Modem	**35**	11	8	8	10	**9**	35	35	11	8	9	11	9	74.3
4pamx mitrec	79	49	48	48	49	51	49	**49**	36	33	35	**35**	35	28.6
block-Vox	472	194	193	193	197	199	409	**409**	138	130	147	**135**	135	67.0
overAddFFT	1476	832	768	768	832	833	1222	**1222**	704	514	514	**514**	577	57.9
phase-dArray	**2496**	2075	2064	2064	**2071**	2072	2496	2496	2076	2064	2064	2071	2072	17.0

It is also interesting to note that of the four possible combinations $((RPMC + sdppo, APGAN + sdppo) \times (ffdur, ffstart))$, the combination of $RPMC + sdppo + ffdur$ gives the best results the most often. Figure 25 shows a bar-graph of the improvement percentage from the last column.

Another experiment was conducted to determine whether applying *ffdur* or *ffstart* on the *sdppo* schedule gives better results than applying *ffdur* or *ffstart* on the *dppo* schedule. The maximum improvement observed on these examples was about 8%. Hence, it is better to use the new DPPO heuristic for shared buffers, although the improvement is not dramatic.

In order to determine whether RPMC and APGAN are generating good topological sorts, we tested the results against the best allocation we could get by generating random topological sorts. We applied the *sdppo* technique, and the firstfit heuristics on this random topological sort to determine the best allocation. For the small graphs like “satrec” and “blockvox” (both with about 25 nodes), we found that it took about 50 random trials to beat the best result generated by the better of RPMC and APGAN-generated schedules. However, even after 1000 trials, the best random schedule resulted in an allocation of 980 for satrec example, and an allocation of 193 for the blockVox example. The best RPMC/APGAN-based allocations are 991 and 199 respectively. So even though we can generate better results just by random search, we cannot improve upon RPMC/APGAN by much, and a lot of time has to be spent doing it.

The situation changes when the random schedules are evaluated for bigger graphs, like the “qmf12_5d” and “qmd235_5d” examples (these have about 200 nodes). Here, after 100 trials, the best allocations were 79 (qmf12_5d) and 8011 (qmf235_5d), compared to 58 and 5690 for the RPMC/APGAN based allocations respectively. Since the running time for 100 trials was already several minutes long on a Pentium II-based PC, we conclude that on bigger graphs, it will require a lot of time and compute power to equal or beat the RPMC/APGAN schedules. Hence, we conclude that APGAN and RPMC are generating topological sorts intelligently, and cannot be easily beaten by non-intelligent strategies such as generating random schedules.

10.2 Homogenous graphs

Unlike previous loop scheduling techniques for buffer memory reduction, the techniques described in this paper are also effective for homogenous SDF graphs. This is because of the allocation techniques;

Fig 25. A bar-graph view of the improvement percentage of the best shared implementation versus the best non-shared implementation.

the sharing strategy can greatly reduce the buffer memory requirement in many cases. As an example, consider the class of homogenous graphs (parametrized by M and N) shown in figure 26. This type of graph (or close variants of it) arises frequently in practice. It is clear that no matter what the schedule is, there are never more than $M + 1$ live tokens. Indeed, running the complete suite of techniques on this graph for any M and N results in an allocation of $M + 1$ units. A non-shared implementation would require $M(N - 1) + 2M$ units instead. The savings are even more dramatic if, along the horizontal chains, vectors or matrices are being exchanged instead of numerical tokens.

10.3 Random graphs

Figure 27 summarizes the results on randomly generated SDF graphs. The experiments were done on 100 random SDF graphs having 20,50,100, and 150 nodes each (400 graphs in all). The chart in figure 27(a) shows the percentage by which the best shared implementation was more than the best non-shared implementation averaged over the 100 graphs for each graph size. As can be seen, the improvement drops off as the graphs get bigger, and is only on the order of 5% for 100 and 150 node graphs. This is in sharp contrast to the performance on practical systems where even the smallest improvements were about 35%. Hence, either random graphs do not appear to show the potential improvement that can result very well, or the random graphs we generate do not correspond very well to practical systems. This is an area that needs further exploration; however, it is hampered by the lack of a large collection of accepted benchmarks of SDF graphs.

Charts in (b) and (c) show how much the allocation differed from the optimistic and pessimistic estimates of the maximum clique weight; as can be seen, the deviation is only on the order of 2-4%. Note that on average, the pessimistic estimate is greater than the allocation (by 1.5-5%) and the optimistic estimate is less than the allocation (by 1.5-4%). The chart in (d) shows the average difference between the best

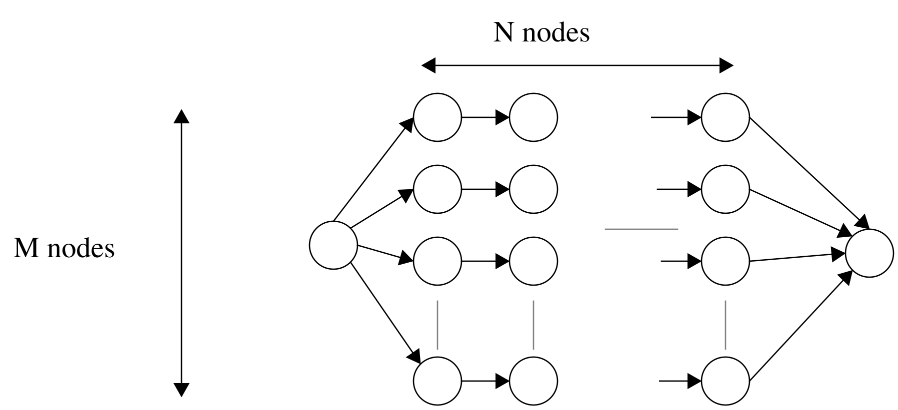


Fig 26. A homogenous graph for which shared allocation techniques are highly beneficial.

allocation and the best *sdppo* estimate. This difference is less than 0.5% on average. Chart (e) shows the difference between RPMC and APGAN; in particular, the average percentage by which the RPMC-generated schedule (post-optimized by *sdppo*, and allocated using the better of *ffdur* and *ffstart*) is better than the APGAN-generated schedule. The difference is significant for larger graphs. Finally, chart (f) shows the

(a) (b)

(c) (d)

(e) (f)

Fig 27. Experimental results on random graphs.

percentage of times the RPMC-based allocation was better than the APGAN-based allocation; this varies from 52-60%. In [3], we observed that RPMC was better than APGAN 66% of the times for non-shared implementations. For shared implementations, the difference between these two is less evident.

11 Related work

11.1 Buffer sharing

11.1.1 Arbitrary schedules (not necessarily SAS)

Bhattacharyya developed a buffer sharing formulation in [4], using lifetime analysis and intersection graph construction, for arbitrary schedules (not necessarily SASs). Since non single appearance schedules can be of exponential length, lifetime analysis and intersection graph construction has much higher complexity in the formulation of [4]. Also, in [4], buffer sharing is not allowed at the fine granularity that is allowed in this paper; for example, periodicity is not taken into account. Finally, no attempt is made in [4] to drive the scheduling algorithm in such a way that the total (shared) allocated memory is minimized.

11.1.2 Sharing on flat single appearance schedules

Ritz et. al. [24] give an approach to minimizing buffer memory that operates only on flat SASs since buffer memory reduction is tertiary to their goal of reducing code size and context-switch overhead (defined roughly as the rate at which the schedule switches between various actors). We do not take context-switch into account in our scheduling techniques because our primary concern is memory minimization as avoiding off-chip memory is often a bottleneck in embedded systems implementations.

Flat SASs have a smaller context switch overhead then nested schedules do, especially if the code-generation strategy used is that of procedure calls. Ritz et. al. formulate the problem of minimizing buffer memory on flat SASs as a non-linear integer programming problem that chooses the appropriate topological sort and proceeds to allocate based on that schedule. This formulation does not lead to any polynomial-time algorithms, and on the satellite receiver example, yields an allocation that is more than 100% of the allocation achieved by techniques developed in this paper. However, the techniques in this paper do not take context-switch overhead into account (since we assume inline code generation, the effect of context switches is arguably less significant), and are thus able to operate on a much larger class of SASs than the class of flat SASs. Also, the techniques in this paper are all provably polynomial-time algorithms.

11.1.3 Buffer minimization by dynamic scheduling

Goddard and Jeffay argue that dynamic scheduling is better suited for reducing memory requirements of SDF graphs, and develop an earliest-deadline-first (EDF) type of dynamic scheduler [10]. They demonstrate their dynamic scheduling approach on the satellite receiver example from [23], and show that the dynamic scheduling approach results in a total buffer memory requirement (assuming that each buffer is implemented separately) of 1599, including the graph input and output buffers—these require 3 tokens with the EDF scheduler [10]. They also do some calculations to show that a shared memory implementation under the EDF scheduler would result in a total requirement of about 1101 locations. Neglecting the issue of input/output buffering for the moment, these requirements are more than that required by static SASs: the non-shared buffer implementation requires 1542 locations, while the shared approach requires 991 locations.

However, the main advantage of dynamic scheduling in this case is the reduction of the **input/output buffering** amount for the graph. The input buffer requirement of a graph (that is to be executed in real-time on a DSP) comes about because the source actors are presumably connected to some outside source that provides data periodically. Similarly, the output actors are connected to the outside world that reads data periodically. These periods can usually be determined based on the sampling rate of the signals that the DSP has to meet. Goddard and Jeffay determine that for the nested, static SAS that APGAN produces for the satellite receiver, 2*1056 tokens would be required for the graph input buffers. Thus, they conclude that the real buffering requirement of the static SAS is 1542+2112 = 3654 tokens. The 1056 comes about because the total number of firings of the source actors A and D is 1056 in one complete period of the SAS. Hence, they assume that since 1056 samples have to be processed by A in a complete period of the SAS, that many samples have to be accumulated in order to execute the SAS correctly. This assumption would be largely correct for a flat SAS since all invocations of actors A and D happen at once. In a nested schedule, however, this is not necessarily true; there might be less time spent executing other actors in a deeply nested loop before the source actor is executed again. Or put another way, the execution of the source actors would be spread out much more over the period of the SAS if the source actors were in nested loops. For instance, the APGAN schedule for the satellite receiver is

$$(24(11(4A)B)CGHI(11(4D)E)FKLM10(NSJTUP))\ (QRV240W).$$

The firing sequence of the initial part of the schedule would be

$$AAAABAAAAB\ldots AAAABCGHIDDDDE\ldots DDDDEFKLMNSJTUP\ldots NSJTUP\ldots.$$

Basically, there are only 44 invocations of A that happen mostly all at once (interspersed with 11 firings of B); there is then a computational delay until another 44 invocations take place and so on. In contrast, a flat SAS would be something like $(1056A)(264B)(24C)\ldots(240W)$. Here the firing sequence would be $A\ldots A$ initially, followed by a long period of time where the other actors are executed. This execution time could potentially span 1056 sample periods; hence, 1056 samples would need to be buffered at the interface to A since A would not be fired all during that time. In the nested schedule, the invocations of A are not spaced out by 1056 sample periods; they happen much more frequently than that. In order to precisely determine how many samples would need to be buffered at the interface to A, we would have to know reasonably accurate execution times of each of the actors in the schedule, and the operating clock speed of the processor that this would be implemented on. We believe that the input buffering required would be considerably less than 1056 samples. We pointed out this observation in [19] for the CD-DAT rate-changing example. We showed that our buffer-optimal SAS (optimized with the non-shared buffer metric) has an input buffering requirement of about 11 tokens, while the flat SAS has a requirement of 65 tokens, assuming some typical execution time values for the actors in the graph, and for a typical DSP in 1994. The total period of the SAS for the CD-DAT example is 147 sample periods. Hence, for the CD-DAT example, the input buffering is far less than 147 (less than 10%), and again, this is because the nesting allows the source actor to fire more frequently than a flat schedule would.

The fact that a non-SAS can have a much smaller buffering requirement is not surprising given the following result from [3]. Consider an edge A, B. Suppose that $prd(\{A, B\}) = a$, $cns(\{A, B\}) = b$, $del(\{A, B\}) = d$, and $gcd(a, b) = c$. Then the minimum buffer size on edge $\{A, B\}$ over all valid schedules is given by $a + b - c + d \bmod c$ if $d < a + b - c$ and d otherwise. The minimum buffer size required over all valid SASs, in contrast, is given by the BMLB [3]: $ab/c + d$ if $d < ab/c$, and d otherwise. Hence, one can devise a greedy, data-driven scheduler that fires a sink actor on an edge in preference to the source actor on that edge whenever both are fireable [3]. For a chain-structured graph, this will result in an optimal schedule in terms of total buffer memory usage. For non-chain-structured graphs, this approach will still result in a buffer memory requirement that is less than the best SAS. This data-driven, greedy scheduler could be implemented as a dynamic scheduler so that we do not pay the penalty of storing a very long schedule that can potentially result. However, it has been shown through experiments in [1] that dynamic scheduling can be up to twice as slow as executing a static schedule. This allows a trade-off: when there are ample cycles to meet throughput demands, but memory is highly restricted, one can use dynamic scheduling so that the least amount of data storage is needed. If throughput demands are stringent, and the overhead of dynamic scheduling is intolerable, one can use SAS optimized for buffer memory usage.

11.1.4 Using n-appearance schedules

Sung et. al consider expanding the SAS to allow 2 or more appearances of some actors if the buffering memory can be reduced [25]. They give heuristic techniques for performing this expansion and show that the buffering can be reduced significantly by allowing an actor to appear more than once. This technique is useful since it allows one to trade-off buffering memory versus code size in a systematic way.

11.2 Improving on single-appearance schedules

SASs will give the least code size only if each actor in the schedule is distinct and has a distinct codeblock that implements its functionality. In reality, however, many actors in the graph will be different instantiations of the same basic actor, with different parameters perhaps. In this case, inline code generated from a SAS is not necessarily code-size optimal since the different instantiations of a single actor could all share the same code [25]. Hence, it might be profitable to implement procedure calls instead of inline code for the various instantiations, so that code can be shared. The procedure call would pass the appropriate parameters. A study of this optimization is done in [25] where the authors formulate precise metrics that can be used to determine the gain or loss from implementing code sharing compared to the overhead of using procedure calls. Clearly, all of the scheduling techniques mentioned in this paper can use this code-sharing technique also, and our work is complimentary to this optimization.

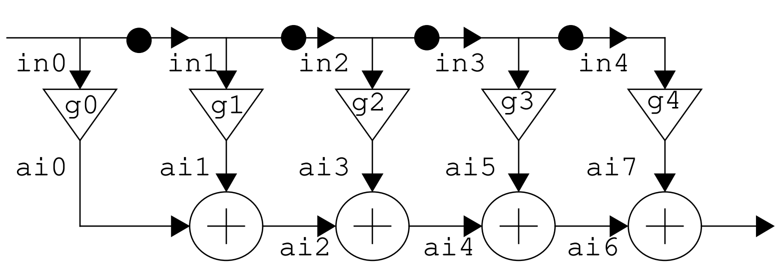
12 Future directions

One of the weaknesses of the buffer sharing model used in this paper is the assumption that all output buffers of an actor are live when the actor begins execution, and all input buffers are live until the actor finishes execution. This means that an output buffer of an actor can never share an input buffer of that actor under the model used in this paper. In reality, this may be an overly restrictive assumption; for instance, an addition actor that adds two quantities will always produce its output after it has consumed its inputs. Hence, the output result can occupy the space occupied by one of the inputs. We have formalized this idea, and have devised another technique called **buffer merging** that merges input and output buffers by algebraically determining precisely how many output tokens are simultaneously live with input tokens (via a formalism called the **consume-before-produce (CBP)** parameter). This technique is highly complimentary to the approach taken by this paper, and is in effect, a dual of the lifetime analysis approach because buffer merging works at the level of a single input/output edge pair, whereas the lifetime analysis approach of this paper works on a global level where the buffering efficiency results from the topology of the graph and the structure of the schedule. The buffer merging technique will be fully presented in a forthcoming paper.

In order to produce code competitive to hand-coded implementations, there are many ways in which additional optimization problems can be formulated. One particular problem that has not been addressed is the issue of recognizing regularity that might occur in a graphical specifications. For instance, consider the fine-grained description of an FIR filter shown in figure 28. The threading approach of inline code generation will generate code as shown in the figure, and clearly, this is not how a programmer would implement this diagram. A hand-coded implementation of this would clearly use a loop to encapsulate the alternating additions and multiplications, and would maintain the gain parameters and input data in arrays that can be indexed by the loop variable.

In [2], we gave a dynamic programming algorithm that can organize loops optimally on a given sequence of actor appearances. If we represent different instantiations of the same actor by the same label, subscripted by instance numbers to differentiate between the various instances, then this algorithm can be used to detect patterns over the labels that can be looped optimally. For instance, if the addition actors are represented by the labels A_i and the gain actors are represented as G_i, then if the schedule generated is $G_0 G_1 A_0 G_2 A_1 \ldots G_n A_{n-1}$, then the algorithm would be able to generate the looped schedule $G_0 n(G_i A_{i-1})$. Clearly, the ability to generate good, compact looped structures would depend on the generated schedule having a regular pattern, and an interesting direction for future study is to study scheduling algorithms that generate schedules that preserve this sort of regularity. Regularity extraction has been applied in the past to high level synthesis [21][22], and Keutzer [12] has applied pattern matching algorithms from compiler design to silicon compilers; perhaps these techniques can be applied in the context of SDF compilers.

In graphical languages, there may be graphical constructs that allow the designer to specify this sort of regularity compactly. A notable example are higher-order functions proposed by Lee [15]. Higher-order functions are graphical blocks that take other graphical blocks as inputs and produce graphical structures as outputs. The graphical structure is produced using simple rules for replicating and connecting the input graphical blocks. Since the regularity has now been captured in a very compact description, the scheduler can make use of this information to preserve this regularity, and not ignore it as is currently done (by only considering the graph that results after all higher-order functions have been statically expanded and removed from the graph). For instance, the "chain" actor is a higher-order function that creates one or




```
mpy ai0,in0,g0
mpy ai1,in1,g1
add ai2,ai0,ai1
mpy ai3,in2,g2
add ai4,ai2,ai3
mpy ai5,in3,g3
add ai6,ai4,ai3
...
```

Fig 28. A SDF specification of an FIR filter at a fine-grained level and the generated code by a threading technique.

more instances of the named actor (it's parameter) connected in a chain. This is implemented by replacing the Chain actor with instances of the named actors statically. The replacement actor(s) are connected as specified by parameters for the Chain actor; these parameters specify how the input edges of the named block map to the input edges of the Chain actor and so on. A parameter called parameter_map determines how parameters are generated for the instantiated actors. Figure 29 shows the graph for constructing the fine-grained FIR description using the Chain actor. It would be interesting to implement scheduling techniques that make use of the information encapsulated in higher-order functions. Certainly, generating large, fine-grained descriptions of algorithms like the FFT or FIR filters is done best by using higher-order functions since other ways of drawing these schematics are usually not scalable, are time-consuming, and are hard-coded for a particular parameter (like the order of the FFT, or the length of the FIR filter). More generally, it would be useful to study scheduling techniques that can make use of the original hierarchy used in the design since large designs are almost always hierarchical.

13 Conclusion

In this paper, we have developed a powerful SDF compiler framework that improves upon our previous efforts demonstrably. By incorporating lifetime analysis into all aspects of scheduling and allocation, the framework is able to generate schedules and allocations that reuse buffer memory, thereby reducing the overall memory usage. In particular, we have developed a model of buffer sharing that we call the coarse-grained model. We gave a new dynamic programming formulation that post-processes flat SASs generated by the APGAN or RPMC algorithms (developed in [3] and [19]) with the shared buffer cost as the cost function to be minimized. We gave several algorithms for extracting the buffer lifetimes from this post-processed SAS; these algorithms are all of polynomial running time in the size of the SAS. We formalized the problem of memory allocation as that of dynamic storage allocation. We have presented extensive experimental results on the performance of this suite of techniques, both on practical systems, and on randomly

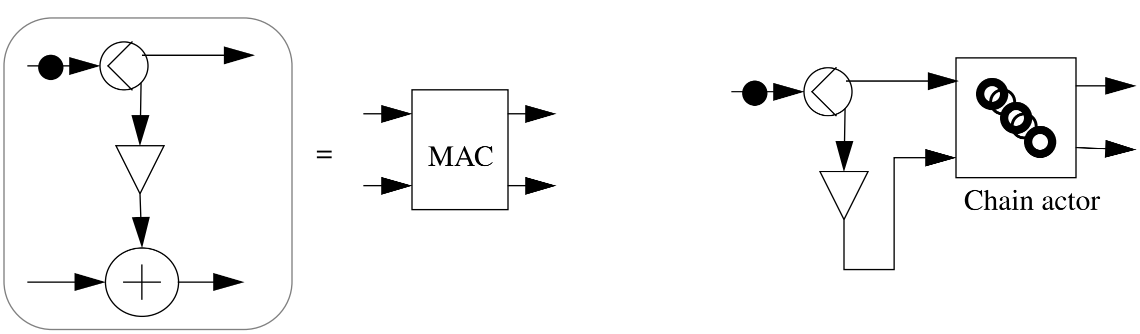


Fig 29. The MAC actor on the left is a hierarchical actor containing the subgraph of the gain and add actors. The Chain actor takes the MAC actor as a parameter and generates the FIR structure shown in figure 28.

generated SDF graphs. We have shown that the improvement over previous techniques, on practical systems, averages more than 50%, with upto 83% improvement on some systems.

14 References

[1] The Almagest, http://ptolemy.eecs.berkeley.edu

[2] S. S. Bhattacharyya, P. K. Murthy, E. A. Lee, “Optimal Parenthesization of Lexical Orderings for DSP Block Diagrams,” Workshop on VLSI Signal processing, Osaka, Japan, 1995.

[3] S. S. Bhattacharyya, P. K. Murthy, E. A. Lee, *Software synthesis from dataflow graphs*, Kluwer Academic Publishers, 1996.

[4] S. S. Bhattacharyya, E. A. Lee, “Memory Management for Dataflow Programming of Multirate Signal Processing Algorithms,” *IEEE Transactions on Signal Processing*, Vol. 42, No. 5, pp. 1190-1201, May 1994.

[5] J. T. Buck, S. Ha, D. G. Messerschmitt, and E. A. Lee, “Multirate signal processing in Ptolemy.” In Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, April 1991, pp. 1245–1248.

[6] J. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, “Ptolemy: a Framework for Simulating and Prototyping Heterogeneous Systems”, *International Journal of Computer Simulation*, Jan. 1995.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1990.

[8] A. Dube, A. Kulkarni, “Benchmarking Code Generation Methodologies for Embedded Programmable DSPs”, final project report, EE382C, Embedded Systems, University of Texas at Austin, May 1997 (http://www.ece.utexas.edu/~bevans/courses/ee382c/projects/spring97/)

[9] M. R. Garey, D. S. Johnson, *Computers and Intractability-A guide to the theory of NP-completeness*, Freeman, 1979.

[10] S. Goddard, K. Jeffay, “Managing Memory Requirements in the Synthesis of Real-Time Systems from Processing Graphs," in Proc. of IEEE Real-Time Technology and Applications Symposium, Denver, Colorado, June 1998.

[11] J. Horstmannshoff, T. Grotker, H. Meyr, “Mapping Multirate Dataflow to Complex {RT} Level Hardware Models,” Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors, pp. 283-293, July 1997.

[12] K. Keutzer, “DAGON: Technology binding and local optimization by DAG matching,” Proceedings of the 24th Design Automation Conference, pp. 617-623, June 1987.

[13] R. Lauwereins, P. Wauters, M. Ade, and J. A. Peperstraete. “Geometric parallelism and cyclo-static data flow in GRAPE-II.” In Proceedings of the IEEE Workshop on Rapid System Prototyping, June 1994.

[14] E. A. Lee, D. G. Messerschmitt, “Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing,” *IEEE Trans. on Computers*, Feb., 1987.

[15] E. A. Lee, T. M. Parks, “Dataflow Process Networks,” *Proceedings of the IEEE*, Vol. 83, No. 5, May 1995.

[16] R. Leupers, P. Marwedel, “Retargetable code generation based on structural processor descriptions,” *Design Automation for Embedded Systems*, Vol.3, No.1, Kluwer Academic Publishers, Jan. 1998. p.75-108.

[17] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang, “Code optimization techniques in embedded DSP microprocessors,” *Design Automation for Embedded Systems*, Vol.3, No.1, Kluwer Academic Publishers, pp. 59-73, Jan. 1998.

[18] P. Marwedel, G. Goossens, *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995.

[19] P. K. Murthy, S. S. Bhattacharyya, E. A. Lee, “Joint Minimization of Code and Data for Synchronous Dataflow Programs,” *Journal on Formal Methods in System Design*, Vol. 11, No. 1, pp. 41-70, July 1997.

[20] P. K. Murthy, S. S. Bhattacharyya, "Approximation Algorithms and Heuristics for the Dynamic Storage Allocation Problem," Tech. Rpt. UMIACS-TR-99-31, University of Maryland Institute for Advanced Computer Studies, College Park, MD 20742, http://www.cs.umd.edu/TRs/TRumiacs.html, 1999.

[21] D. S. Rao, F. J. Kurdahi, "An Approach to Scheduling and Allocation Using Regularity Extraction," Proceedings of the European Conference on Design Automation, pp. 557-561, 1993.

[22] D. S. Rao, F. J. Kurdahi, "On Clustering for Maximal Regularity Extraction," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, Vol. 12, No. 8, pp. 1198-1208, August 1993.

[23] S. Ritz, M. Pankert, and H. Meyr. "Optimum vectorization of scalable synchronous dataflow graphs." In *Proceedings of the International Conference on Application Specific Array Processors*, October 1993.

[24] S. Ritz, M. Willems, H. Meyr, "Scheduling for Optimum Data Memory Compaction in Block Diagram Oriented Software Synthesis," Proceedings of the ICASSP 95, Detroit, Michigan, May 1995.

[25] W. Sung, J. Kim, S. Ha, "Memory efficient synthesis from dataflow graphs," International Symposium on System Synthesis, Hinschu, Taiwan, 1998.

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

[27] M. C. Williamson, "Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph Specifications," Ph.D. Thesis, Memorandum No. UCB/ERL M98/45, Electronics Research Laboratory, University of California at Berkeley, June 1998.

[28] V. Zivojinovic, J. M. Velarde, C. Schlager, H. Meyr, "DSPStone — A DSP-oriented Benchmarking Methodology," International Conference on Signal processing Applications and Technology, 1994.