Shared Buffer Implementations of Signal Processing Systems Using Lifetime Analysis Techniques

Praveen K. Murthy, *Member, IEEE,* and Shuvra S. Bhattacharyya, *Member, IEEE*

***Abstract*—There has been a proliferation of block-diagram environments for specifying and prototyping digital signal processing (DSP) systems. These include tools from academia such as Ptolemy and commercial tools such as DSPCanvas from Angeles Design Systems, signal processing work system (SPW) from Cadence, 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) has been found to be a particularly good match for expressing multirate signal processing systems. One of the key problems that arises during synthesis from an SDF specification is scheduling. Past work on scheduling from SDF has focused on optimization of program memory and buffer memory under a model that did not exploit sharing opportunities. In this paper, we build on our previously developed analysis and optimization framework for looped schedules to formally tackle the problem of generating optimally compact schedules for SDF graphs. We develop techniques for computing these optimally compact schedules in a manner that also attempt to minimize buffering memory under the assumption that buffers will be shared. This results 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 well-known 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 up to an 83% improvement over previous techniques.**

***Index Terms*—Block diagram compiler, DSP software synthesis, dynamic programming, dynamic storage allocation, lifetime analysis, loop fusion, memory allocation, static scheduling, synchronous dataflow, weighted interval graph coloring.**

I. Introduction

BLOCK diagram environments are proving to be increasingly popular for developing digital signal processing (DSP). 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 and 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 overspecifying 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 (HLL) like $C{+}{+}$. For software synthesis, the technique typically used is that of inline code generation: 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 or the code may be hand-optimized code targeted for a particular target implementation. 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. This latter technique is generally more efficient for programmable DSPs because of a lack of efficient HLL DSP compilers.

For hardware synthesis, a similar approach can be taken, with blocks implementing their functionality in a hardware description language, like behavioral VHSIC hardware descripton language (VHDL) [12], [34]. The generated VHDL description can then be used by a behavioral synthesis tools to generate a register transfer level (RTL) description of the system that can be further compiled into hardware using logic synthesis and layout tools.

HLL compilers for DSPs have been woefully inadequate in the past [35]. 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 Texas Instruments Incorporated, their newest DSP architecture, is a very large instruction work (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

Manuscript received May 1, 2000; revised October 1, 2000. S. Bhattacharyya was supported in part by the U.S. National Science Foundation under Contract 9734275. This paper was recommended by Associate Editor R. Camposano.

P. K. Murthy is with Angeles Design Systems, San Jose, CA 95113 USA (e-mail: pmurthy@angeles.com).

S. S. Bhattacharyya is with the University of Maryland, College Park, MD 20742 USA (e-mail: ssb@eng.umd.edu).

Publisher Item Identifier S 0278-0070(01)00940-X.

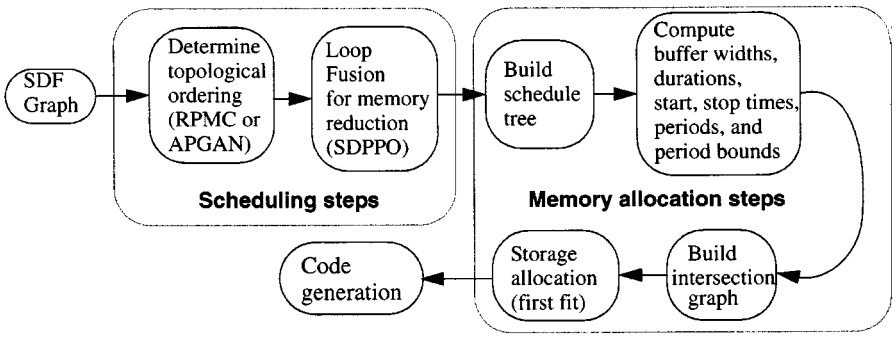

Fig. 1. Flowchart showing the sequence in which the various algorithms are applied.

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 (e.g., see [19]–[21]).

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 synthesis tool 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 compiler for a general-purpose HLL (such as C) typically does not have the global information about application structure that our scheduler has. The techniques we develop in this paper are, thus, complementary to the work that is being done on developing better HLL compilers for DSPs such as that presented in [19]–[21]. 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 HLL like C. If the latter, then we would have to invoke a C compiler after performing the dataflow 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 [15].

II. Problem Statement and Organization of the Paper

In this paper, we describe a technique for reducing buffering requirements in synchronous dataflow (SDF) graphs based on lifetime analysis and memory allocation heuristics for single-appearance looped schedules (SAS). As already mentioned, the first step in compiling SDF graphs is determining a schedule. Once the schedule has been determined, memory has to be allocated for the buffers in the graph. Both of these steps present many algorithmic challenges; we tackle many of these steps in this paper. We concentrate on the class of SASs in our framework because nonSASs for SDF graphs can be exponentially long; this can lead to very large code size [4]. Within the class of SAS, there are two algorithmic challenges: to determine the order in which the actors should appear in the schedule, subject to the precedence constraints imposed by the graph (the topological ordering), and the order in which the loops should be organized once the order has been determined. Solutions to both of these problems depend on the optimization metric of interest. In this paper, the metric is buffer memory; hence, these algorithms all try to minimize the amount of buffer memory needed. While previous techniques for buffer minimization have used techniques where each buffer is allocated independently in memory (we will refer to this as the *nonshared model*), in this paper we try to share buffers efficiently by using lifetime analysis techniques (referred to as the *shared model*). In the memory allocation steps, the challenges are to efficiently extract buffer lifetimes from the schedule and to pack these buffers into memory efficiently. All of the algorithms we present are provably polynomial-time algorithms; this is important because SDF compilers are often used in rapid-prototyping environments where fast compile times are necessary and desirable.

In Section III, we review relevant past work on this subject. Sections IV and V establish some of the notation and definitions we will use. Fig. 1 summarizes the various algorithms that we will develop in this paper as part of our SDF compiler framework. The box with "RPMC or APGAN" in Fig. 1 finds the topological ordering and is reviewed in Section VII (briefly, since these algorithms have been developed previously). The box with "SDPPO" solves the loop-ordering problem and is described in Section VII. After the SDPPO step, we will have a theoretical idea of the amount of buffer memory required, but as will be shown, until the actual memory allocation is performed, we do not know the exact requirements. The memory allocation steps take the schedule produced by the first two steps and attempt to determine the most efficient allocation. In order to do this, we have to build a tree representation of the schedule; this is covered in Section V. On this representation, several parameters that are needed for lifetime analysis, like the start and stop times of buffers, their periodicities, and durations have to be computed; algorithms for doing this efficiently are given in

Section VIII. Once all of the lifetime parameters have been determined, another structure called an intersection graph has to be built. On this graph, allocation heuristics are applied in order to get a final memory allocation; these two steps are covered in Section IX. Since we have developed two efficient heuristics for generating topological orderings in the scheduling step, neither of which can be said to be clearly superior and since we have developed two heuristics that can be used in the memory allocation step, our experiments in Section X examine all four of the possible combinations to determine the most efficient combination for each test example. In Section XI, we discuss possibilities for future work and conclude.

III. Related Work

Lifetime analysis techniques for sharing memory are well known in a number of contexts. The first is for register allocation in traditional compilers; given a scheduled dataflow graph, register allocation techniques determine whether the variables in the graph can be shared by looking at their lifetimes. In the simplest form, this problem can be formulated as an interval graph coloring problem that has an elegant polynomial-time solution. However, the problem of scheduling the graph so that the overall register requirement is minimized is an NP-hard problem [30]. Register allocation problems are made somewhat simpler because the variables in question all have the same size. The allocation problem becomes NP-complete if variables are of differing sizes, as for example, in allocating arrays of different sizes to memory.

Fabri [8] studies the more general problem of overlaying arrays and strings in imperative languages. Fabri models array lifetimes as weighted interval graphs and uses coloring heuristics for generating memory allocations. She also studies transformation techniques for lowering the overall memory cost; these techniques attempt to minimize the lower and upper bounds on the extended chromatic number of the weighted interval graph. Some transformation techniques found to be effective for reducing overall storage include: the renaming transformation, whereby with the use of judicious renaming of aggregate variables, lifetimes can be fragmented, which allows greater opportunities for overlaying; the technique of recalculation, where some variables are recalculated when needed rather than holding them in storage; code-motion techniques that reorder the program statements in a semantics preserving manner; and loop splitting.

There are important differences between Fabri's work and ours. Fabri considers general imperative language code, and hence has to solve allocation problems for a more general class of interval graphs. We apply our techniques on SDF graphs and because the SDF model of computation is restricted, the interval graphs in our problem have a more restricted structure, enabling us to use simpler allocation heuristics more effectively. For instance, the liveness profile of an array in our framework is always periodic (in a certain technical sense) and these periods can be deduced from the SDF graph and the specific class of schedules that we use, whereas in a general setting, liveness profiles may not be periodic and deducing these profiles can be expensive algorithmically. Also, the SDF model and SDF schedules present unique problems for deducing the liveness profiles and, thus, the interval graphs in an efficient manner; these techniques have not been presented or studied in any previous work. We show that for the important class of SASs, these deductions can be made in polynomial time in the size of the SDF graph. We present an optimization technique for reducing the extended chromatic number by performing loop fusion in a systematic manner. While the loop fusion technique is applicable in a general setting as well, opportunities for doing it in a general setting do not arise as frequently and naturally as they do in an SDF setting, hence, it is a very effective technique here. For example, determining the applicability of loop fusion is undecidable in procedural languages, whereas exact analysis is decidable and tractable in our context. Thus, loop fusion is more effective for SDF graphs and our work exploits this increased effectiveness. Also, previous work has not addressed the relationship been loop fusion and the extended chromatic number. Finally, even though certain subsets of the techniques we present in this paper have been studied in the compilers community, to date they have not been used in block-diagram compilers. An additional contribution of this paper is to show that many of the techniques used in traditional compilers can be specialized and applied fruitfully in block-diagram-based DSP programming environments.

Vanhoof *et al.* [33] have observed that in general, the full address space of an array does not always contain live data. Thus, they define an "address reference window" as the maximum distance between any two live data elements throughout the lifetime of an array and fold multiple array elements into a single window element using a modulo operation in the address calculation. This concept is similar to our use of the maximum number of live tokens as the size of each individual SDF buffer. The number of logically distinct memory elements in a buffer for an edge e is equal to $TNSE(e)$, which can be much larger than the maximum number of live tokens that reside on e simultaneously [4].

In a synthesis tool called ATOMIUM, De Greef *et al.* [11] have developed lifetime analysis and memory allocation techniques for single-assignment static control-flow specifications that involve explicit looping constructs such as *for* loops. This is in contrast to SDF in which all iteration is specified implicitly and the use of looping is left entirely up to the compiler. However, once a single-appearance schedule is specified, we have a set of nested loops. Thus, some relationships can be observed between the lifetime analysis techniques we develop for SASs and those of ATOMIMUM. In particular, the class of specifications addressed by ATOMIMUM exhibits more general and less predictable array-accessing behavior than the buffer access patterns that emerge from SDF-based SASs. We exploit the increased predictability of SASs in our work using novel lifetime analysis formulations that are derived from a tree-based schedule representation. This results in thorough optimization with significantly more efficient (lower complexity) algorithms. Furthermore, through our in-depth focus on the restricted but useful class of SDF-based SASs, we expose fundamental relationships between scheduling and buffer sharing in multirate signal processing systems.

Ritz *et al.* [29] 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; off-chip memory is often a bottleneck in embedded systems implementations and is better avoided.

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 nonlinear 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 can lead to much more expensive memory allocations than those obtainable through nested schedules. For example, in Section X, we show that on a satellite receiver example, Ritz's technique yields an allocation that is more than 100% larger than 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.

Goddard and Jeffay use a dynamic scheduling strategy for reducing memory requirements of SDF graphs and develop an earliest-deadline-first (EDF) type of dynamic scheduler [10]. However, experiments in the Ptolemy system have shown that dynamic scheduling can be more than twice as slow as static schedules [36]. Hence, for many embedded applications, this penalty on throughput might be intolerable.

Sung *et al.* consider expanding the SAS to allow two or more appearances of some actors if the buffering memory can be reduced [31]. 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 tradeoff buffering memory versus code size in a systematic way.

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 an SAS is not necessarily code-size optimal since the different instantiations of a single actor could all share the same code [31]. 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 [31] 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; our work is complementary to this optimization.

Ade [1] has developed lower bounds on memory requirements of SDF specifications, assuming that each buffer is assigned to separate storage. Exploring the incorporation of buffer sharing opportunities into this analysis is a useful direction for further investigation.

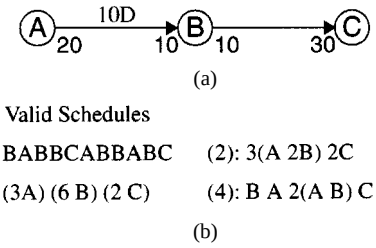

Fig. 2. (a) Example of an SDF graph. (b) Some valid schedules.

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 first-in first-out (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 SDF [17]. 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.

IV. Notation and Background

Fig. 2(a) shows a simple SDF graph. Each edge is annotated with the number of tokens produced (consumed) by its source (sink) actor and the "10 D" on the edge from actor A to actor B specifies 10 delays. Each unit of delay is implemented as an initial token on the edge. Given an SDF edge e, we denote the *source* actor (that writes tokens on the edge), *sink* actor (that reads tokens from the edge), and *delay* of e by $src(e)$, $snk(e)$, and $del(e)$. Also, $prod(e)$ and $cns(e)$ 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 $prod(e) = cns(e)$ 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 or procedure call 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 [4], 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 min-

imum 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

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

The vector q, when it exists, is called the *repetitions vector* of G. A *schedule* is then a sequence of actor firings where is each actor v is fired $q[v]$ times and the firing sequence obeys the precedence constraints imposed by the SDF graph. For the graph is Fig. 2, we have $q = [3, 6, 2]$ for the actors $[A, B, C]$, and some schedules are $BABBCABBABC$, $AAABBBBBBCC$, and $BAAABBCBBBC$.

We define $TNSE(e)$ to be the total number of samples exchanged on edge e by actor $snk(e)$; i.e., $TNSE(e) = q[snk(e)] \cdot cns(e)$.

V. Constructing Memory-Efficient Loop Structures

In [4], the concept and motivation behind 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. If the SAS restriction is removed, significant increase in code size can occur. The increase in code size will manifest itself even if inline code generation is not used and subroutine calls are used instead. This is because the length of a non-SAS can be exponential in the size of the graph, and there could be exponentially many subroutine calls.

Fig. 2(b) shows some valid schedules for the graph in Fig. 2(a). The notation $2C$ represents the firing sequence CC. Similarly, $3(A(2B))$ represents the schedule loop with firing sequence $ABBABBABB$. We say that the iteration count of this loop is three and the body of this loop is $A(2B)$. Schedules 2 and 3 in Fig. 2(b) are SASs since actors A, B, C appear only once. An SAS like the third one in Fig. 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 [4].

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 Fig. 2(b), the buffering requirements for the four schedules, assuming that one separate buffer is implemented for each edge, are 50, 90, 130, and 80, respectively. As can be seen, SASs can have significantly higher buffer requirements than a schedule optimized purely for buffer memory. For example, the non-SAS $BABBCABBABC$ for the SDF graph of Fig. 2 has a buffer requirement of 50; the three possible SASs for the graph $3A6B2C$, $3(A2B)2C$, and $3A2(3BC)$, have requirements of 130, 90, and 100, respectively. We give priority to code-size minimization over buffer memory minimization; justification for this may be found in [4] and [24]. 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.

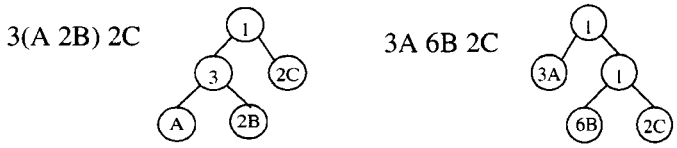

Fig. 3. Schedule trees for schedules (2) and (3) in Fig. 2(b).

A. R Schedules and the Schedule Tree

In order to extract buffer lifetimes efficiently, we develop a useful representation of the nested SAS, called the schedule tree. The lifetime extraction algorithms of Section VIII can then be formulated as tree-traversing algorithms for determining the various required parameters.

As shown in [24], it is always possible to represent any SAS for an acyclic graph as

$$(i_L S_l)(i_R S_R) \tag{1}$$

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 iteration counts 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 (1). SASs having this form (in conjunction with some additional technical restrictions on the loop iteration counts) at all levels of the loop hierarchy are called R schedules [24].

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 iteration count of the subschedule rooted at that node. The *leaf nodes* (nodes that have no children) will contain the actors, along with their residual iteration counts. If a node v has children, we refer to the *left child* and *right child* of v by $left(v)$ and $right(v)$. For a node u, the *parent* is referred to as $parent(u)$. Fig. 3 shows schedule trees for the SASs in Fig. 2(b). Note that a schedule tree is not unique since if there are iteration counts of one, then the split into left and right subgraphs can be made at multiple places. In Fig. 3, the schedule tree for the flat SAS in Fig. 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.

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 $loop$: $V \rightarrow Z$, where V is the set of nodes in the tree, and Z is the set of positive integers, returns for a nonleaf node, the iteration count at that nesting level and returns one for a leaf node.

VI. Generating Single Appearance Schedules

We have shown [4] that for an arbitrary acyclic graph, an 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 $(q_1x_1)(q_2x_2)\cdots(q_nx_n)$, where the x_i are actors and q_i are the repetitions $q[x_i]$. The lexical order $x_1x_2\cdots x_n$ is the order given by the topological sort of the graph). This lexical ordering then leads to a set of nesting

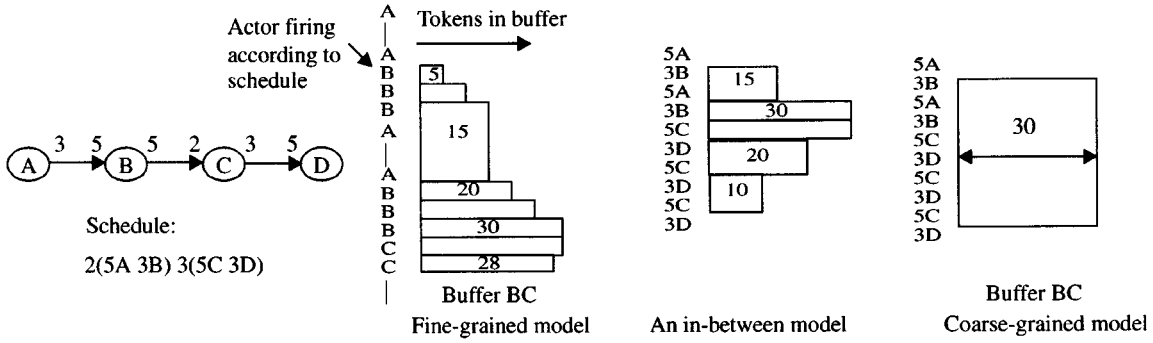

Fig. 4. Fine-grained, in between, and coarse-grained models of buffer sharing.

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 we have shown [4], the general problem of constructing buffer-optimal SASs under both models of buffering, namely, the coarse shared buffer model and the nonshared model are NP-complete. Thus, the methods for generating topological sorts are necessarily heuristic and not optimal in general.

We have developed two methods for generating SAS optimized for nonshared buffer memory for acyclic graphs [4]: a bottom-up method based on clustering called acyclic pairwise grouping of adjacent nodes (APGAN) and a top-down method based on graph partitioning called recursive partitioning by minimum cuts (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 nonshared buffer model, it is also attractive for the shared model as will be shown.

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 nonshared buffer memory metric over all SAS [4].

An arbitrary SDF graph may not necessarily have an SAS; Bhattacharyya *et al.* [5] developed necessary and sufficient conditions for the existence of SAS for SDF graphs. They developed an algorithm for generating SASs that hierarchically decomposes the SDF graph into strongly connected components (SCC) and recursively schedules each SCC. At each stage, the SCC decomposition results in an acyclic component graph that has an SAS as mentioned and can be scheduled using any algorithm for generating SAS for acyclic graphs. Hence, the techniques we develop in this paper can all be incorporated into the framework of [5] and can handle arbitrary SDF graphs.

VII. Efficient Loop Fusion for Minimizing Buffer Memory

Once we have a topological order generated by APGAN or RPMC, we have a flat SAS corresponding to this topological order. The next step is to perform loop fusion on the flat SAS to reduce buffering memory. To do that, we first define the shared buffer model.

A. The Shared Buffer Model

Since we are interested in sharing buffers, we have to first determine an appropriate model for buffer lifetimes and the manner in which they can be shared. First, we need a definition for describing token traffic on the edges: given an SDF graph G, a valid schedule S, and an edge e in G, let $max\_tokens(e, S)$ denote the maximum number of tokens that are queued on e during an execution of S. For example, if for Fig. 2, $S_3 = (3A)(6B)(2C)$ and $S_2 = 3(A\,2B)(2C)$, then $max\_tokens((A, B), S_3) = 70$ and $max\_tokens((A, B), S_2) = 30$.

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 actor 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 $max\_tokens(e, S)$ tokens immediately become live and stay live until the number of tokens on the edge becomes zero, where S is the schedule. In other words, even if there is one live token on the edge, we assume that an array of size $max\_tokens(e, S)$ has to be allocated and maintained until there are no live tokens. Fig. 4 shows these two extremes pictorially for the buffer on edge BC. In the fine-grained case, each firing of B results in the buffer on BC expanding by five and each firing of C results in the buffer contracting by two. In the coarse-grained case, the buffer expands to 30 immediately as all six firings of B are treated as one composite firing and then shrinks to zero after all 15 firings of C have occurred. Of course, there are a number of granularities within these extremes based on how many levels of loop nests we consider; Fig. 4 shows the in between alternative for this example, where only the outer loop of iteration count two is considered, meaning that the three firings of B in the inner loop are treated as one composite firing. The buffer, in this case, expands by $3 \times 5 = 15$ tokens on each composite firing of B and contracts by $5 \times 2 = 10$ on each composite firing of C consisting of five firings.

In this paper, we assume the coarsest level of buffer modeling. The finer levels, although requiring less memory theo-

retically, may be practically infeasible to achieve due to the increased complexity of the algorithms. To see that the complexity might significantly increase, notice that the finest level requires modeling to be done at the granularity of single firing of an actor in the schedule. The number of firings in a periodic SDF schedule is $O(P^m)$, where $P = \mathrm{MAX}_{e \in E}\{prod(e), cns(e)\}$, E is the set of edges in the SDF graph, and $m = |E|$. Of course, there may be more clever ways of representing the growth and shrinkage, but presently the only known ways are equivalent to stepping through a schedule of size $O(P^m)$. Clearly, this is an exponential function in the size of the SDF graph and can grow quickly. In contrast, we will show that the coarsest level model can be generated in time polynomial in the number of nodes and edges in the SDF graph.

One weakness of the coarse buffer sharing model 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* [23], [37] 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 *consumed-before-produced* (CBP) parameter [3]). The buffer merging technique is similar in spirit to the array merging technique presented by DeGreef *et al.* [11]; however, it is more efficient in some ways and also exploits distinguishing characteristics of SDF schedules in a novel way. We have shown that the buffer merging technique is highly complementary to the approach taken in this paper and is, in effect, a dual of the lifetime analysis approach. This is 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 [22], [23], [37].

Initial tokens on edges can be handled very naturally in our coarse shared buffer 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 duration 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.

In order to reason about the "start time" and "stop 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 $2(A3B)$ would be considered to take four time steps. This is because the firing sequence is $A3B\,A3B$ and since the schedule loop $3B$ is a leaf node in the schedule tree, it is considered to take one schedule step. The first invocation of A would take place at time zero and the last invocation of $3B$ begins at time three and ends at time four. Note that this notion of time is not used to judge run-time performance of the schedule in terms of throughput; it is simply used to define the lifetimes for purposes of lifetime analysis.

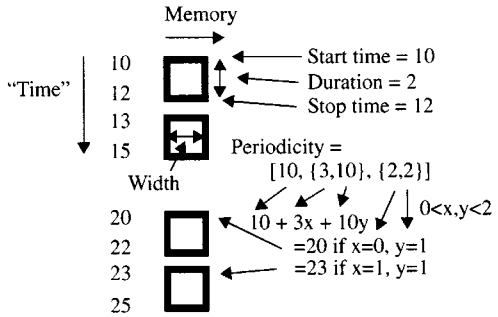

Fig. 5. Anatomy of a buffer lifetime.

Fig. 5 shows the anatomy of a buffer lifetime. Notice that this buffer becomes live several times. The start time is defined as the very first time the buffer becomes live; in this contrived example, at time ten. The stop time is defined as the very first time the buffer stops being live—at time 12 in the example. The duration is simply the difference between the stop and start times. The periodicity is modeled by a three-tuple as shown; this will be described in greater detail in Section VIII-D. Briefly, it is modeled by a Diophantine equation so that the start time of the ith occurrence of the buffer can be computed algebraically. Finally, the width of the buffer is defined to be $max\_tokens(e, S)$ as mentioned already for the coarse-grained model.

B. Loop Fusion Under the Nonshared Buffer Model—DPPO

Once the topological ordering of the nodes in the SAS has been determined, we have to determine the order in which loops should be nested since, as shown in Section V and Fig. 2, the loop hierarchy has significant impact on buffer memory usage. In [4] and [24], we developed a postprocessing technique based on dynamic programming [called dynamic programming post optimization (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 because the technique we describe for generating good loop hierarchies under the shared model is similar.

For the nonshared model, we define the nonshared buffer memory requirement of a schedule S by

$$nsbufmem(S) \equiv \sum max\_tokens(e, S) \qquad (2)$$

where the summation is over all edges in G. Thus, $nsbufmem(S_3) = 70 + 60 = 130$, and $nsbufmem(S_2) = 30 + 60 = 90$ in Fig. 2.

The lexical ordering of an 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 an SAS that has minimum nonshared buffer memory requirement from among the valid SASs that have a given lexical ordering.

One of the central observations that allows the development of an efficient algorithm for optimizing buffer memory under

the nonshared model can be stated intuitively in the following way: fusing adjacent loops together by taking out their common factor (and creating an outer loop that has the common factor as its iteration count) not only gives us a valid schedule but also gives us a schedule that has a nonshared buffering memory requirement that is equal or smaller than the nonshared buffering memory requirement of the schedule with the set of separate loops. Formally, we can state it as [24]:

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

1) S' is a valid schedule for G;
2) $nsbufmem(S') \leq bufmem(S)$. [with $nsbufmem$ as defined by (2)].

The schedule loop L' is called the factored loop. The act of factoring out a common factor like γ is called loop fusion or factoring the schedule.

Informally, the DPPO algorithm uses a divide-and-conquer approach that looks at a chain of actors in the schedule and examines each point in the chain to determine the buffer cost of breaking the chain there and fusing the "left" and "right" parts. It then picks the best point to split the chain at and records it and considers bigger chains. Because this problem can be shown to have the "optimal subproblem" property, meaning that optimal solutions for the "left" and "right" parts lead to an optimal solution to the whole chain, DPPO is an optimal algorithm for the nonshared buffer model [24].

Formally, 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_{00} is an order-optimal schedule for $(G, lexorder(S))$. If G contains at least two actors, then it can be shown [24] that there exists a valid schedule of the form $S_R = (i_LB_L)(i_RB_R)$ such that $nsbufmem(S_R) = nsbufmem(S_{00})$ 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_{00}, (i_LB_L) and (i_RB_R) are also order-optimal (otherwise, we can show that we could replace (i_LB_L) or (i_RB_R) by order equivalent versions without affecting the split costs).

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) $(b-a) \leq (n-2)$ and 2) $a = 1$ or $b = n$. Given these schedules, an order-optimal schedule for G can be derived from a value of x, $1 \leq x < n$ that minimizes

$$nsbufmem(S_{1,x}) + nsbufmem(S_{x+1,n}) + \sum_{e \in E} TNSE(e)$$

where

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

is the set of edges that "cross the split" if the schedule 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 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 [24].

C. Loop Fusion for the Shared Buffer Model–SDPPO

Now that we have a model for sharing buffers, we develop an algorithm for organizing loops efficiently in an SAS such that the shared buffer cost is minimized. This algorithm is similar to the DPPO algorithm already described. Consider the following dynamic programming formulation:

$$bufmem(S_{1,n}) = \text{MIN}_{1 \leq x \leq n} \left\{ \text{MAX}\,\{bufmem(S_{1,x}), bufmem(S_{x+1,n})\} + \sum_{e \in E_s} TNSE(e) \right\} \quad (3)$$

where the last term (sum of $TNSE$) is again the sum of the buffer costs crossing the cut. The term $bufmem(S)$ is the shared buffer memory requirement, and (3) serves as the definition. Of course, $bufmem(S) = 0$ if S has only one actor. The maximum of the left and right costs $(bufmem(S_{1,x})$ and $bufmem(S_{x+1,n}))$ is taken based on the intuition shown in Fig. 6. Since the buffers in the subgraph on the right side of the cut 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 (which can be live simultaneously with both the left and right set of buffers). The right buffers are overlayed with the left ones.

The formulation in (3) 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 Fig. 7. For the given schedule, the top-level partition occurs on edge DE, with a cost of 36. According to the formulation, the total cost

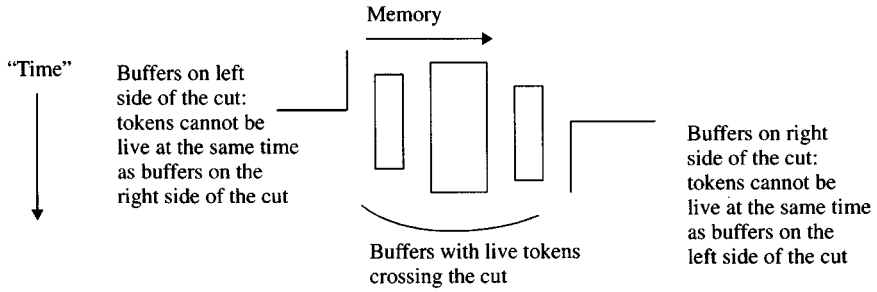

Fig. 6. Intuition for a revised dynamic programming formulation.

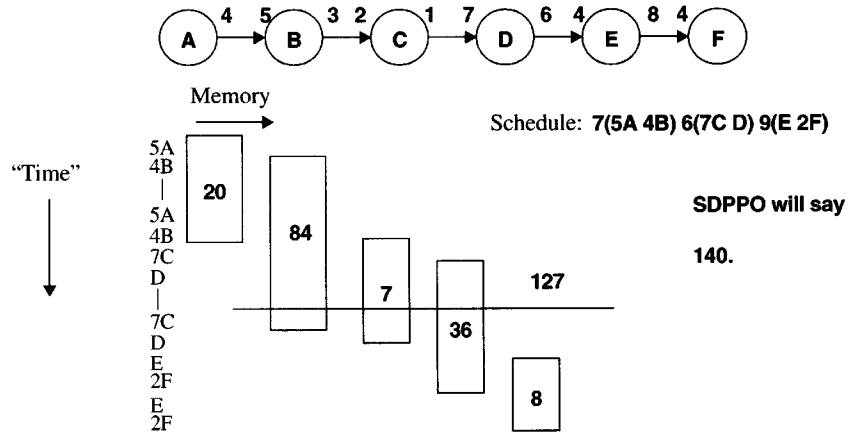

Fig. 7. An example to illustrate that the shared DPPO is suboptimal.

should be $6 + \max(b[A \cdots D], b[E \cdots F])$, where $b[A \cdots D]$ is the buffering cost of the subschedule consisting of actors A, B, C, D. For $b[A \cdots D]$, the partition occurs on edge BC. Hence, the cost is given by $84 + \max(20, 7) = 104$. For $b[E \cdots F]$, we have $b[EF] = 8$. Hence, the total cost gets computed as $36 + \max(104, 8) = 140$, while the actual cost is only 127 as shown by Fig. 7.

Fact 1 says that loop fusion never increases buffer memory usage under the nonshared model; unfortunately, this is not true under the shared model. Indeed, consider the example in Fig. 8. In Fig. 8(a) and (b), there is no edge between actors A and B. If we do not perform loop fusion, as shown in Fig. 8(a), we see that buffer profiles will be such that the buffers on the output edges of actor B will be disjoint from the buffers on the input edges of actor A. If we perform loop fusion, as shown in Fig. 8(b), these buffers will no longer be disjoint, thus preventing sharing. Moreover, no advantage is gained from the fusion since the sizes of the input and output buffers do not decrease; loop fusion 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 Fig. 8(c) and (d), then loop fusion 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 of (3): we do not perform loop fusion if there are no internal edges (that is, edges whose terminal points are all actors that are being merged).

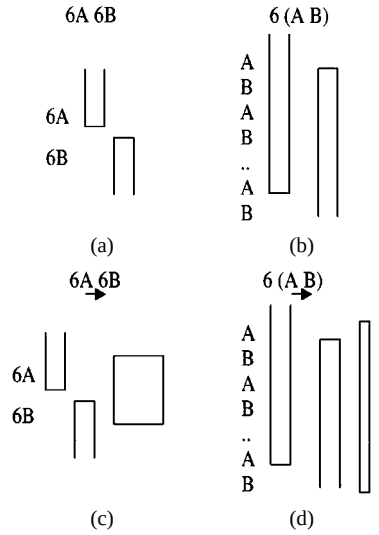

Fig. 8. Example to illustrate that factoring can increase buffering requirement under the shared model.

We perform loop fusion 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 increase the complexity of the algorithm. We choose to use the simpler approach of using the heuristic approach to determine whether to fuse or not, and leave for future work to explore more complex approaches. We define this formulation of DPPO [(3) together with the heuristic of deciding when

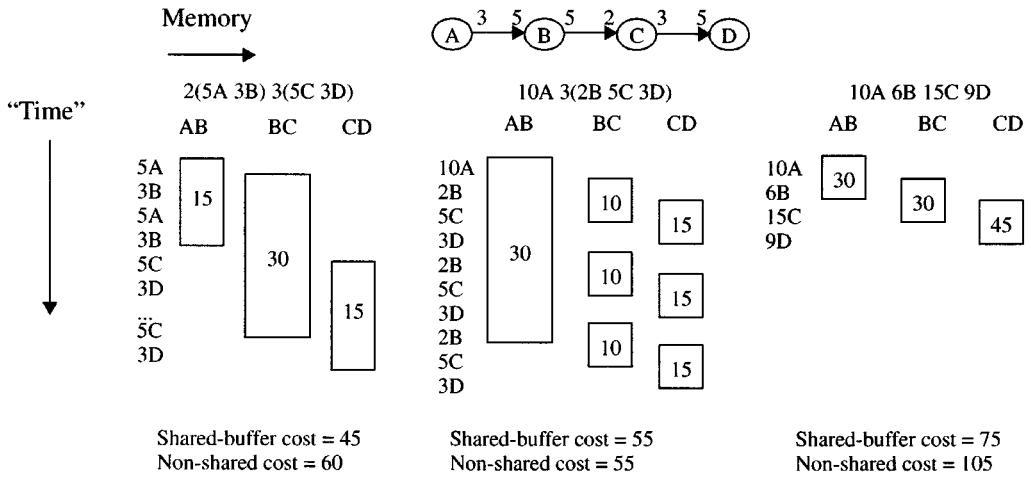

Fig. 9. Example to show that the shared-buffer optimal schedule is not the same as the nonshared-buffer optimal schedule.

to perform loop fusion] as shared dynamic programming post optimization (SDPPO).

Note that the best schedule under the nonshared buffering model is not necessarily the best schedule under the shared model as shown in Fig. 9. Finally, we can also see that RPMC is an attractive heuristic for the shared buffer model since the cut-crossing buffers will not be disjoint, and cannot be shared. Hence, it makes intuitive sense to drive the partitioning process by minimizing the size of these buffers and this is what RPMC attempts to do.

VIII. Creating the Interval Instances from an SAS

Once the schedule and the loop hierarchy have been determined, the next step in the compilation process is to perform memory allocation. Even though the SDPPO algorithm gives us a number for the overall memory requirement, it is only an estimate since the SDPPO algorithm cannot determine whether that estimate can actually be achieved. The main difficulty is that packing a number of arrays of different sizes optimally is an NP-complete problem, hence, the optimal amount of memory required after packing cannot be determined until the packing has actually been performed.

The two main steps for memory allocation are to extract the buffer lifetimes, and then perform allocation using those lifetimes. Extracting the lifetimes efficiently requires several algorithms for determining the durations and the start and stop times. 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. Hence, the lifetime extraction algorithms also have to model this periodicity efficiently. Given the lifetimes, the allocation step (packing arrays in other words) determines the physical location in memory, where the buffer will reside.

We compute these parameters for the buffers from the schedule tree. However, note that a parameter computed for a buffer is a function of a pair of actors (that constitute the edge that the buffer is on). The schedule tree does not represent these edges directly, only the actors and structure of the nested loops. Hence, we first compute these parameters for nodes in the schedule tree; these parameters will represent the start time, stop time, and durations of the various nested loops. We can then deduce the buffer lifetimes from the computed quantities for the nested loops.

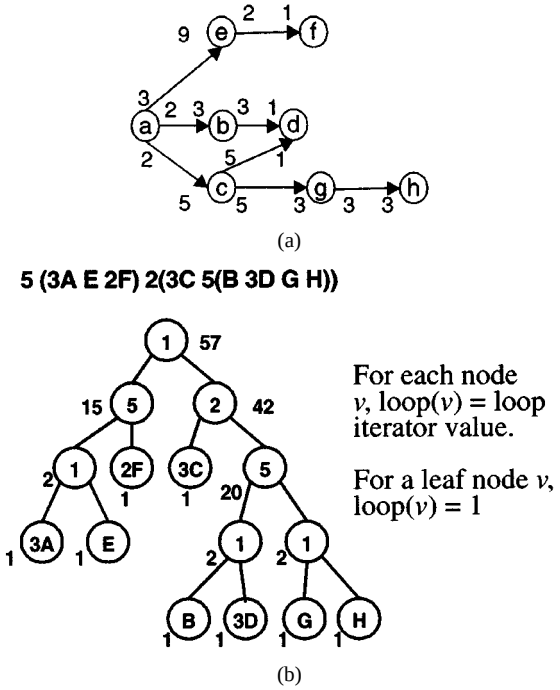

Fig. 10. (a) SDF graph. (b) Binary tree representation of an SAS for the graph in (a), along with the duration values for the nodes in the schedule tree.

We will use a running example to show these various algorithms; the example SDF graph is depicted in Fig. 10(a), with an SAS.

A. Computing the Duration Times of Loop Nests

The function *loop* (defined in Section V-A) is used in the computation of the duration times $dur(v)$ for all nodes v (i.e., loop nests) in the schedule tree by depth-first search on the tree

$$dur(v) = loop(v)(dur(left(v)) + dur(right(v))) \qquad (4)$$

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

```
Procedure computeStartStopTimes(scheduleTree) {
      doComp(root,0)
}
Procedure doComp(v, start) {
      start(v) ← start
      stop(v) ← start + dur(v)
      if (v not leaf node)
            doComp(left(v), start);
            doComp(right(v), stop(left(v)))
      fi
}
```

Fig. 11. Pseudocode for the depth-first search algorithm for computing start and stop times for all the nested loops.

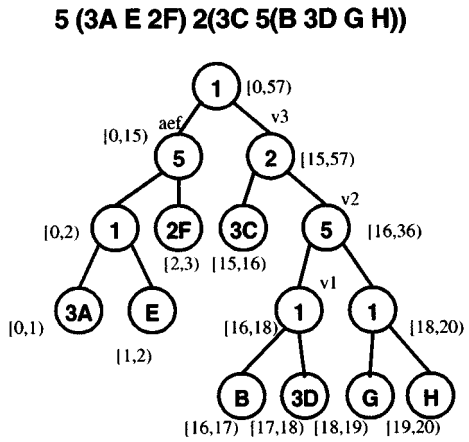

Fig. 12. Start and stop times for each node computed using depth-first search.

Fig. 10(b) show the result of the depth-first search computation of the duration values. The depth first search takes time $O(|V|)$.

B. Computing the Start and Stop Times of Loop Nests

The next task is to compute the start and stop times for each nested loop. These times are defined as

$$start(v) = \begin{cases} start(parent(v)) & v \text{ is a left child} \\ stop(left(parent(v))) & v \text{ is a right child} \end{cases}$$
$$start(root) = 0, stop(v) = start(v) + dur(v).$$

These times can also be computed using depth-first search [taking time $O(|V|)$], as shown by the pseudocode in Fig. 11. The SAS from Fig. 10(b) is shown with the computed start and stop times in Fig. 12. The start time of a nested loop represents the first time the loop nest starts execution. The stop time represents the first time the loop nest finishes execution. For example, consider the edge (A, B) in Fig. 10(a). The stop time computed for leaf node B in the schedule tree is 17. This corresponds to the first time B finishes execution: five firings of $3A\ E\ 2F$ correspond to 15 "steps" in the measuring scheme described in Section VII-A (hence, the stop time computed for the node marked "aef" in Fig. 12 is Fig. 15), and after this there is a firing of $3C$, and B, giving 17 as the stop time for the first execution of B as shown in Fig. 12.

C. Computing the Start, Stop, and Durations for Buffer Lifetimes

Now we have to compute the lifetimes for the buffers from the parameters we have computed for the loop nests. We introduce some notation for nodes in the schedule tree first.

Definition 1: A *common ancestor* of a pair of nodes u_1, u_2 is any node a that contains the nodes u_1, u_2 as leaf nodes in the subtree rooted at a.

Definition 2: The *least common ancestor* of a pair of nodes u_1, u_2 is the *first* node a (measured from the leaf nodes) that contains nodes u_1, u_2 as leaf nodes in the subtree rooted at a.

Definition 3: The *greatest common ancestor* of a pair of nodes u_1, u_2 is the last node a that contains nodes u_1, u_2 as leaf nodes in the subtree rooted at a such that all ancestors of a have a loop value of unity.

Definition 4: The *common ancestor set* of a pair of nodes u_1, u_2 is the set of all common ancestors of u_1, u_2 on the path from the least common ancestor to the greatest common ancestor.

The least and greatest common ancestors of a pair of leaf nodes correspond to the innermost and outermost loops that contain the actors corresponding to the leaf nodes. In Fig. 12, the common ancestor set for the leaf node pair $\{B, 3D\}$ is $v1$, $v2$, $v3$. 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 since the firing of actor u makes the buffer on edge (u, v) live. The stop time of the buffer interval is the first time it 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; this is because the stop time computed for the leaf node represents the first time the corresponding sink actor finishes execution. However, the first time the sink actor finishes execution is not necessarily the first time all tokens in the buffer would have been consumed. For instance, consider the edge (A, B) in Fig. 10(a). The stop time computed for leaf node B in the schedule tree is 17. However, notice that buffer AB will not stop being live until all ten firings of B have occurred.

Hence, we really need to compute the time when the last execution of the sink actor v of the buffer takes place in the loop nest of interest. The loop nest of interest is the smallest loop nest containing both u and v since the total number of tokens consumed by v in this loop nest has to equal the number produced by u in that loop nest. However, the stop time computed for the node representing the smallest loop nest (that is, the least common ancestor in the schedule tree) includes the execution of all loop nests contained within it. We want to exclude the contribution to the stop time from all nests that follow the sink actor in the last execution of the loop nest of interest. Again, using the example of Fig. 12, the loop nest of interest for buffer AB is the root node since that is the least common ancestor. The stop time of 57 computed for the root node includes the execution $3D$ and G in the final execution of the loop nest corresponding to the root node. We want to subtract the execution times of $3D$ and G from 57 in order to get the time at which B finishes execution for the

```
Procedure computeIntervalStopTime

foreach buffer(u, v)
      //find the least common ancestor of (u,v) in scheduleTree
      leastCommonAncestor ← findLeastCommonAncestor(u, v)
      stop_uv ← stop(leastCommonAncestor)
      tmp ← vleaf  // vleaf = leaf node corresponding to actor v
      while (tmp ≠ right(leastCommonAncestor))
            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. 13. Procedure for computing the earliest stop time of an interval.

last time (i.e., $57-2=55$). This idea is formalized in the algorithm shown in Fig. 13; it takes $O(|V||E|)$ time. The algorithm first finds the least common ancestor in the schedule tree for u, v using the procedure *findLeastCommonAncestor* [done by computing the ancestor sets of each leaf node (u, v) and then determining the first common ancestor from those sets; takes time $O(|V|)$]. The stop time of the buffer interval for (u, v) is set to the stop time computed for the least common ancestor. The algorithm then moves up toward the least common ancestor from leaf node v and adjusts the stop time according to the move: if the move up is from the left, then the duration of the right node (representing the execution of loop nests following v) is subtracted; if the move up is from the right, no adjustment is necessary. The algorithm stops when it reaches the right child of the least common ancestor.

D. 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 Fig. 14. 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 Fig. 14 for buffers on edges (B, D) and (G, H).

For a buffer b on an (U, V) edge in the SDF graph, let the common ancestor set be denoted as nodes $v_1^b, \ldots, v_n^b$, where v_1^b is the least common ancestor, v_2^b is the next ancestor, and so on. For example, for the buffer on edge (B, D) in Fig. 12, the least and greatest common ancestors are the nodes marked $v1$ and $v3$ for the leaf-node pair $\{B, D\}$. The node $v2$ is also in the common ancestor 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

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

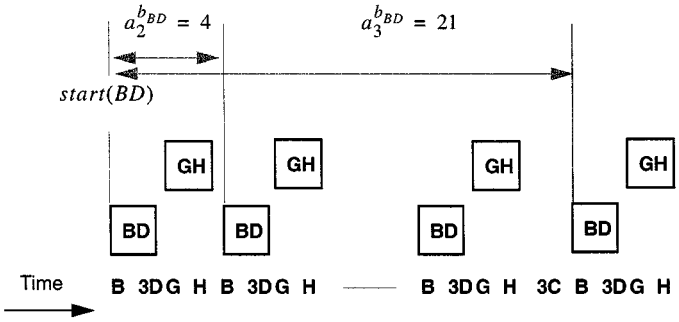

Fig. 14. Two periodic intervals corresponding to the schedule from Fig. 12.

where a_i^b are the constants $dur(left(v_i^b)) + dur(right(v_i^b))$. Nodes in the common ancestor set that have *loop* values of unity will not contribute to the periodicity (shown in the example below); hence, those components can be eliminated. So n is the size of the common ancestor set minus the number of common ancestors that have loop values of unity. This triple allows us to represent the buffer lifetime in the following way: buffer b is live for time intervals

$$\begin{aligned}&\left[start(b) + p_1 a_1^b + \cdots + p_n a_n^b,\right.\\ &\quad \left. start(b) + p_1 a_1^b + \cdots + p_n a_n^b + dur(b)\right]\end{aligned}$$

for all combinations of $p_i \in \{0, \ldots, loop(v_i^b) - 1\}$. If $loop(v_i^b)$ is unity for any common ancestor, then that common ancestor does not contribute to the periodicity since zero is the only value p_i can take.

For example, for the buffer BD in Fig. 14, which we denote b_{BD}, we have that $start(b_{BD}) = 16$, $dur(b_{BD}) = 2$, $(a_1^{b_{BD}}, a_2^{b_{BD}}, a_3^{b_{BD}}) = (2, 4, 21)$ (corresponding to the common ancestor set of $v1, v2, v3$) and $\{loop(v1), loop(v2), loop(v3)\} = \{1, 5, 2\}$. We can remove the first component as the loop value is unity. Hence, buffer BD is live during the intervals given by $[16 + 4y + 21z,\ 16 + 4y + 21z + 2]$ for all combinations of $y \in \{0, 1, 2, 3, 4\}$, $z \in \{0, 1\}$. Some of the live intervals are [16, 18], [20, 22], and [37, 39].

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

$$start(b) + p_1 a_1^b + \cdots + p_n a_n^b \leq T \leq start(b) p_1 a_1^b + \cdots + p_n a_n^b + dur(b) \quad (5)$$

where the p_i are variables that range over $\{0, \ldots, loop(v_i^b) - 1\}$ has a solution in p_1. A solution, if it exists, can be found easily because of the following property on the p_i. Let the a_i^b be sorted in increasing order. Then the following must hold.

Lemma 1:

$$\sum_{i=1}^{j} a_i^b \left(loop\left(v_i^b\right) - 1\right) \leq a_{j+1}^b \qquad j = 1, \ldots, n-1.$$

Intuitively, the lemma states that the start time of a buffer due to the $i+1$th outer loop has to be after all occurrences of start times for the buffer have taken place due to the ith inner loop and all loops contained in the ith loop. This is intuitively true because since the loops are nested, the outer loop count increments only after the inner loops have counted through their entire range.

Proof: (by induction on j). Since $dur(v_i^b) = loop(v_i^b)(dur(left(v_i^b)) + dur(right(v_i^b)))$, and $dur(v) \geq 1$, $loop(v) \geq 1$ for all nodes v in the schedule tree, we have that for the parent v_{i+1}^b of v_i^b,

$$\begin{aligned} dur\left(v_{i+1}^b\right) &= loop\left(v_{i+1}^b\right)\left(dur\left(v_i^b\right) + dur\left(right\left(v_{i+1}^b\right)\right)\right) \\ &\geq dur\left(v_i^b\right) \end{aligned}$$

where we have assumed, without loss in generality, that $left(v_{i+1}^b) = v_i^b$. Now, since

$$a_i^b = dur\left(left\left(v_i^b\right)\right) + dur\left(right\left(v_i^b\right)\right)$$

we have $dur(v_i^b) = a_i^b loop(v_i^b)$. For $j = 1$, we need to show that $a_1^b(loop(v_1^b) - 1) \leq a_2^b$. Indeed

$$\begin{aligned} a_1^b\left(loop\left(v_1^b\right) - 1\right) &= dur\left(v_1^b\right) - a_1^b \leq dur\left(v_1^b\right) \\ &\leq dur\left(v_1^b\right) + dur\left(right\left(v_2^b\right)\right) = a_2^b. \end{aligned}$$

Now assume that the claim holds for $j-1$. To prove it for j, we have

$$\begin{aligned} &\sum_{i=1}^{j} a_i^b \left(loop\left(v_i^b\right) - 1\right) \\ &\quad = \sum_{i=1}^{j-1} a_i^b \left(loop\left(v_i^b\right) - 1\right) + a_j^b \left(loop\left(v_j^b\right) - 1\right) \\ &\quad \leq a_j^b + a_j^b \left(loop\left(v_j^b\right) - 1\right) = a_j^b loop\left(v_j^b\right) \\ &\quad = dur\left(v_j^b\right) \leq dur\left(v_j^b\right) + dur\left(right\left(v_{j+1}^b\right)\right) = a_{j+1}^b. \end{aligned}$$

Q.E.D.

Corresponding to the sorted n-tuple $\overline{a}_b = [a_1^b, \ldots, a_n^b]$, define the tuple $\overline{p} = [p_1, \ldots, p_n]$, where the ith component is p_i. Note that p_i are only allowed to be in the range $0 \leq p_i < loop(v_i^b)$. Define an ordering relationship $\overline{p}_c < \overline{p}_d$ in the following way. Let i be the largest index such that the ith components of $\overline{p}_c$ and $\overline{p}_d$ differ. Then $\overline{p}_c < \overline{p}_d$ iff the ith component of $\overline{p}_c$ is less than the ith component of $\overline{p}_d$. Define the start time of an occurrence of the periodic buffer by the function $s(\overline{p}_b) = \overline{p}_b \cdot \overline{a}_b^T$, where $(\cdot)^T$ denotes the vector transpose. Then we have the following lemma.

Lemma 2: $\overline{p}_c < \overline{p}_d \Leftrightarrow s(\overline{p}_c) < s(\overline{p}_d)$ for any tuples $\overline{p}_c$, $\overline{p}_d$, with $0 \leq p_i^c, p_i^d < loop(v_i^b)\ \forall i$.

Proof: ($\Rightarrow$ direction): Since $\overline{p}_c < \overline{p}_d$, we have for the largest i where the two tuples differ $p_i^c < p_i^d$. Since the last $n - i$ components of $\overline{p}_c$ and $\overline{p}_d$ are the same, we have

$$\begin{aligned} s(\overline{p}_d) - s(\overline{p}_c) &= a_1^b(p_1^d - p_1^c) + \cdots + a_{i-1}^b(p_{i-1}^d - p_{i-1}^c) \\ &\quad + a_i^b(p_i^d - p_i^c) \\ &\geq -\sum_{k=1}^{i-1} a_k^b(loop(v_k^b) - 1) + a_i^b(p_i^d - p_i^c) \geq 0 \end{aligned}$$

since $p_i^d - p_i^c > 0$ and by using Lemma 1.

($\Leftarrow$ direction): Again, letting i be the highest index where $\overline{p}_c$ and $\overline{p}_d$ differ, we need to show that $p_i^c < p_i^d$. Suppose not. Suppose $p_i^c > p_i^d$ but $s(\overline{p}_c) < s(\overline{p}_d)$. We have

$$\begin{aligned} 0 &< s(\overline{p}_d) - s(\overline{p}_c) \\ &= a_1^b(p_1^d - p_1^c) + \cdots + a_{i-1}^b(p_{i-1}^d - p_{i-1}^c) + a_i^b(p_i^d - p_i^c) \\ &\leq a_1^b p_1^d + \cdots + a_{i-1}^b p_{i-1}^d - a_i^b < 0 \end{aligned}$$

by Lemma 1 again, contradicting our assumption that $p_i^c > p_i^d$. **Q.E.D.**

1) An algorithm for computing buffer liveness at a particular time: Given Lemma 1, (5) can be solved by the algorithm in Fig. 15. The algorithm first subtracts the start time of the buffer since all computations can be made relative to the start time. It then simply determines the maximum $p_i^b \in \{0, \ldots, loop(v_i^b) - 1\}$ factor for each a_i^b, to determine the closest starting point of an occurrence of the periodic interval to time T.

Claim 1: $T \geq 0$ at every stage in the algorithm.

Proof: Note that $p_i \leq \lfloor T/a_i^b \rfloor$ for all i. Hence $T - p_i a_i^b \geq T - \lfloor T/a_i^b \rfloor a_i^b \geq 0$. **Q.E.D.**

Claim 2: The solution p_i computed by the algorithm gives the starting point of the interval closest to T starting before T.

Proof: Let $\overline{p}$ be the tuple consisting of the p_i. This means that any tuple $\overline{p}' > \overline{p}$ gives an interval of starting time greater than T. Indeed, suppose not and there is a tuple $\overline{p}' > \overline{p}$ where $s(\overline{p}) < s(\overline{p}') < T$. Then, $p_j' > p_j$ for the largest index j, where the two tuples differ. Until the jth step, the value of T computed by the algorithm would be identical to the computation using the values from $\overline{p}'$. When $i = j$, the algorithm computes $p_j = \min(\lfloor T/a_j^b \rfloor, loop(v_j^b) - 1)$. Suppose $p_j = \lfloor T/a_j^b \rfloor$. Clearly, anything larger than p_j will mean that $T - p_j a_j^b < 0$, giving a start time greater than T. If $p_j = loop(v_j^b) - 1$, then we cannot have $p_j' > p_j$ since $loop(v_j^b) - 1$ is the largest value the jth component is allowed to take by definition. Hence, we cannot have $p_j' > p_j$, contradicting the assumption that $\overline{p}' > \overline{p}$. **Q.E.D.**

The last step of the algorithm checks whether $T < dur(b)$ to determine whether the interval with closest starting time less than or equal to T is still alive.

If b is not live at time T, we will need to determine when the next instance of its periodic interval will occur. This computation is needed to determine whether some other interval of a particular duration is completely disjoint with the set of

```
Given: a time T. Determine if a periodic buffer b
parametrized by {start(b), (a_1^b, ..., a_n^b), (loop(v_1^b), ..., loop(v_n^b))}
is live at this time.

Define T' ← T − start(b).
for i = n to 1 step -1
        p_i ← min( ⌊ T' / a_i^b ⌋ , loop(v_i^b) − 1 )
        T' ← T' − p_i a_i^b
end for
if ( T' < dur(b) ), then b is live (p_i are the solution)
else not live.
```

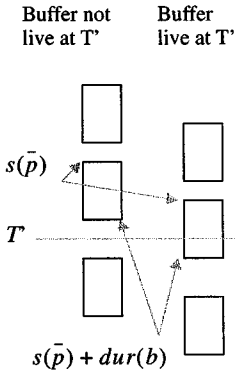

Fig. 15. Algorithm to determine whether a periodic buffer is live at a particular time. Depiction on the right shows two possibilities for the computed start time by the algorithm in relation to T'.

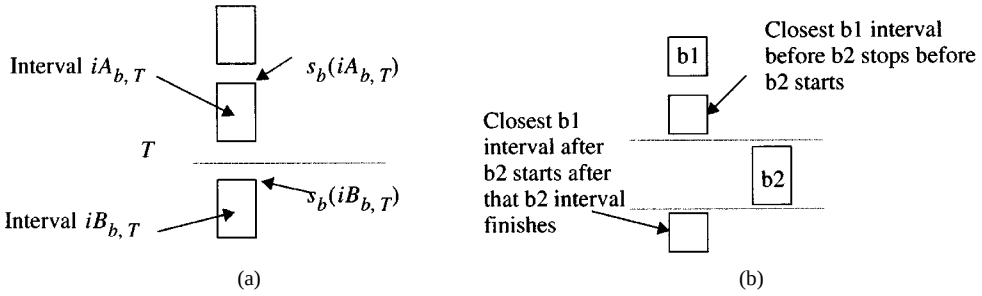

Fig. 16. (a) Nearest intervals before and after time T. (b) Condition for nonintersection of two periodic intervals from Lemma 3 depicted graphically.

intervals corresponding to b; that is, to 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 p_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 (p_i) be (0, 1, 1). The number this represents is $0 \cdot 28 + 1 \cdot 13 + 1 \cdot 4 = 17$. 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.

These ideas can be formalized in the following lemma. For a periodic interval b, let $s_b(i)$, $e_b(i)$ be the start and stop times of the ith occurrence of the interval. That is, $s_b(i) = start(b) + \overline{p}_b^i \cdot \overline{a}_b^T$ for the ith increment of the "number" p_j^i, $j = 1, \ldots, n$ and $e_b(i) = s_b(i) + dur(b)$. Given a time T, let $iA_{b,T}$ be the interval of b nearest to T with start time less than or equal to T. That is, $\forall\, s_b(i) \leq T$, $s_b(i) \leq s_b(iA_{b,T}) \leq T$. Similarly, $iB_{b,T}$ is defined as the nearest interval of b with start time greater than T. Fig. 16(a) illustrates these definitions.

Lemma 3: Two periodic intervals b_1, b_2, with $start(b_2) > start(b_2) > start(b_1)$ do not intersect if and only if

$$e_{b_1}(iA_{b_1,start(b_2)}) \leq start(b_2)$$

and

$$s_{b_1}(iB_{b_1,start(b_2)}) \geq start(b_2) + dur(b_2). \qquad (6)$$

In other words, the two intervals do not intersect if and only if the closest interval of b_1 that starts before the start time of b_2 finishes before the start time of b_2 AND the closest interval of b_1 that starts after the start time of b_2 starts after that interval of b_2 finishes [Fig. 16(b)]. Notice that the lemma says that we do not have to consider other occurrences of the periodic interval b_2 to determine overlap—only the first occurrence.

Proof: The forward direction is trivially true. The reverse direction can be established via a case analysis. Let the two edges on which buffers b_1, b_2 reside be given by (u_{b_1}, v_{b_1}) and (u_{b_2}, v_{b_2}). Since $start(b_2) > start(b_1)$, the ordering of these actors in the schedule must be one of $u_{b_1} v_{b_1} u_{b_2} v_{b_2}$, $u_{b_1} u_{b_2} v_{b_1} v_{b_2}$, or $u_{b_1} u_{b_2} v_{b_2} v_{b_1}$. Clearly, the condition of (6) cannot be satisfied for the third order since b_1 is live the entire time that b_2 is. For the other two orders, we have to consider the different ways in which the loops could be nested. For each order, there are five distinct ways of nesting the loops. These five are, for the first order, the following: $I_1 u_{b_1} I_2(I_3(I_4 v_{b_1} I_5 u_{b_2}) I_6 v_{b_2})$, $I_1 u_{b_1} I_2(I_3 v_{b_1} I_4(I_5 u_{b_2} I_6 v_{b_2}))$, $I_1(I_2 u_{b_1} I_3 v_{b_1}) I_4(I_5 u_{b_2} I_6 v_{b_2})$, $I_1(I_2(I_3 u_{b_1} I_4 v_{b_1}) I_5 u_{b_2}) I_6 v_{b_2}$, and $I_1(I_2 u_{b_1} I_3(I_4 v_{b_1}) I_5 u_{b_2})) I_6 v_{b_2}$. Note that we only consider the part of the overall schedule that contains these four actors (the subtree of the schedule tree rooted at the least common ancestor of these four nodes) and we ignore any other actors that appear in the order or nesting as they do not affect the properties of the particular buffers we are interested in. Fig. 17 shows the buffer profiles for these five cases. As can be verified, (6) holds if and only if the intervals do not intersect. We can similarly verify that the lemma is true for the five nestings for the other order. **Q.E.D.**

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

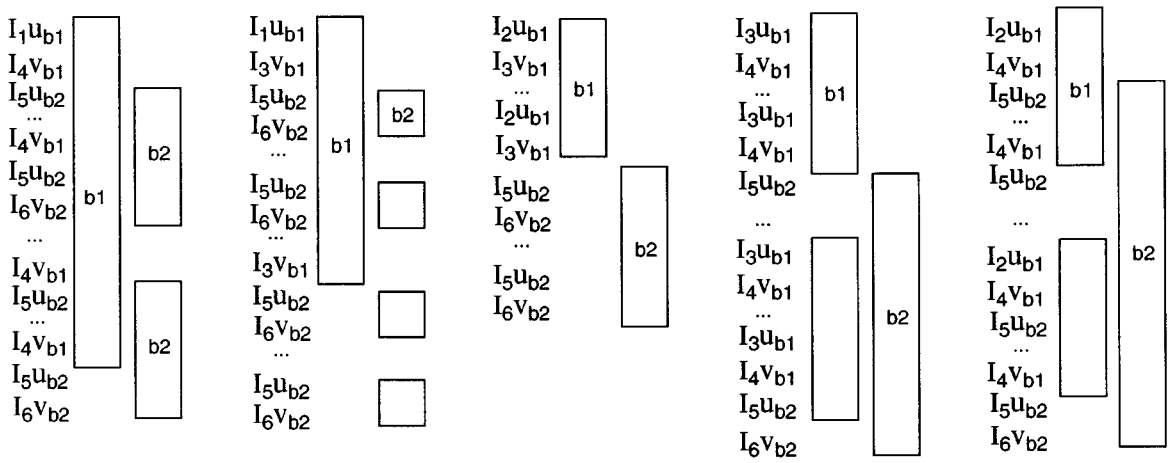

Fig. 17. Buffer profiles for the five possible nesting hierarchies in Lemma 3.

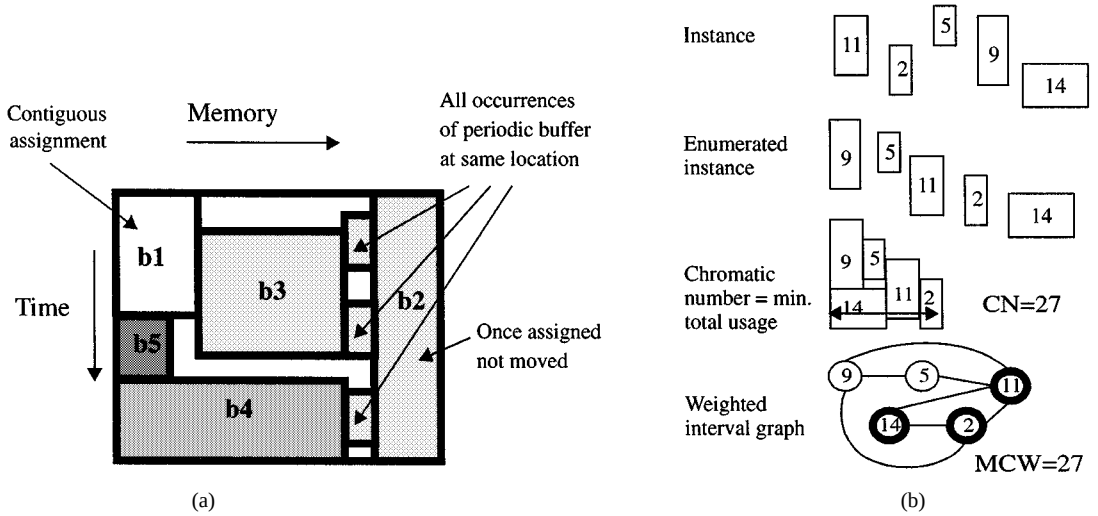

Fig. 18. (a) Memory allocation properties. (b) DSA terminology.

$O(|V|)$ 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 common ancestor set of $O(|V|)$ nodes for any buffer between actors in the innermost loop. Hence, in the procedure in Fig. 15, $n = O(|V|)$, 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.

IX. 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 the problem is to arrange the different sized arrays in memory so that the total memory required during any time is minimized. The assignment to memory is assumed to have the following properties: 1) an array is assigned to a contiguous block of memory; 2) once assigned, an array may not be moved around; and 3) all occurrences of an array with a periodic lifetime profile are assigned to the same location in memory. Fig. 18(a) depicts these properties. Of course, we could relax any of these restrictions and perhaps get smaller memory requirements but it might come at the expense of other overheads [like moving arrays around if (2) were relaxed]. We leave to future work to investigate these other models for allocation. Formally, DSA is defined as the following.

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: $B \rightarrow \{0, \ldots, K-1\}$ such that $o \leq A(b) \leq K - w(b)$ 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 $A(b_1) + w(b_1) \leq a(b_2)$ or $A(b_2) + w(b_2) \leq A(b_1)$.

The "dynamic" in DSA refers to the fact that many times, the problem is on line 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 sizes 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 one and two.

A. 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. Fig. 18(b) shows these definitions via an example.

B. Heuristic for DSA

First fit (FF) is the well-known algorithm that performs allocation for an enumerated instance by assigning the smallest feasible location to each interval in the order they appear in the enumerated instance [13]. It does not reallocate intervals that have been allocated already, and it does not consider intervals not yet assigned. The pseudocode for this algorithm is shown in Fig. 19. We refer the reader to a technical report [25] and references therein for a more detailed treatment of this very interesting DSA problem. Briefly, the algorithm takes as input an enumerated instance. We tested two types of orderings for generating enumerated instances [25]: ordering by start times, and ordering by durations. It then builds the WIG using the routine `buildIntersectionGraph`. The WIG is built using the general test developed for determining intersection of possibly periodic buffers. The FF algorithm then examines the WIG for each buffer i: first it examines all nodes adjacent to i in the WIG (i.e., buffers that intersect i). It collects the memory allocations of all the adjacent nodes that appear before i in the enumeration. After sorting these allocations, it sees where i can be allocated; in the worst case, it has to be allocated at the end of all of the allocations because there are no regions big enough in between to accommodate i. After an allocation is determined for i, the next buffer is examined in the enumeration until all have been allocated.

Our study shows that in practice, FF is a good heuristic, and we use it in our compiler framework here. Our empirical study on random WIGs shows that ordering the buffers by durations gives the better results [25]. But, in our experiments in Section X, we will apply FF on both ordering by start times (abbreviated *ffstart*), and ordering by durations (*ffdur*).

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 and the 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.

C. Computing the Maximum Clique Weight

It is clear that the maximum clique weight is a lower bound on the chromatic number of a weighted interval graph. It is known that the chromatic number can be as much as 1.25 times the maximum clique weight for particular instances; however, it is not known whether 1.25 is a tight upper bound. The maximum clique weight 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 [25] show that *ffdur* comes within 7% on average of the maximum clique weight, in practice, the chromatic number is not much bigger than the maximum clique weight, certainly not as much as 1.25 times as big. Hence, we use the maximum clique weight for comparison purposes in our experiments in the next section.

While the maximum clique weight can be computed easily and exactly for an instance without fragmented lifetimes, computing it for instances with fragmented (but periodic) buffer lifetimes is more difficult. Consider the case where all intervals are continuous (i.e., not fragmented). 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 CW. 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 Fig. 20). Hence, to compute the MCW in this scenario, we would have to consider start times of all occurrences of a periodic interval; this becomes a nonpolynomial 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 algorithm of Fig. 15. This is an optimistic estimate since the MCW 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).

```
Procedure FirstFit(enumerated instance I)

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

foreach buffer 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
      // neighborsAllocations contains memory addresses
      sort(neighborsAllocations) //by increasing memory address
      foreach allocation a ∈ neighborsAllocations
            if allocate[i] conflicts with a
                  //w(a) = size of interval with allocation a
                  allocate[i] ← a + w(a)
            fi
      end for
end for


Procedure buildIntersectionGraph(enumerated instance I)

sort I by start times (or durations for ffdur)
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) < stop 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 FF heuristic.

X. Experimental Results

We have tested these algorithms on several practical benchmark examples, as well as on random graphs. As mentioned earlier, the crux of the experiment is to study the memory requirement as a result of using the best combination of the four possibilities

$$(RPMC + sdppo, APGAN + sdppo) \times (ffdur, ffstart).$$

That is, perform the scheduling by using one of RPMC or APGAN to generate the topological ordering, and perform loop fusion on that schedule using SDPPO. Then, perform memory allocation using one of *ffstart* (FF with buffers ordered by starting times) or *ffdur* (FF with buffers ordered by durations). We compare the best memory requirement obtained this way to the best memory requirement from nonshared techniques, namely, applying one of RPMC or APGAN and loop fusion using DPPO (note that when buffers are not shared, the memory allocation step is trivial since each buffer gets a separate block of memory). Fig. 21 shows the percentage improvement on 16 systems we tested. As can be seen, there is, on average, more than a 50% improvement by using the compiler framework of this paper compared to previous techniques. On some examples, the improvement is as high as 83%. Details of the experiments are given below.

A. Practical Multirate Systems

The practical multirate examples are a number of one-sided filter bank structures [32] as shown in Fig. 22, two-sided filter banks [32], as shown in Fig. 23, and a satellite receiver example [29] as shown in Fig. 24. Another type of variation that occurs frequently in practical signal processing systems is vari-

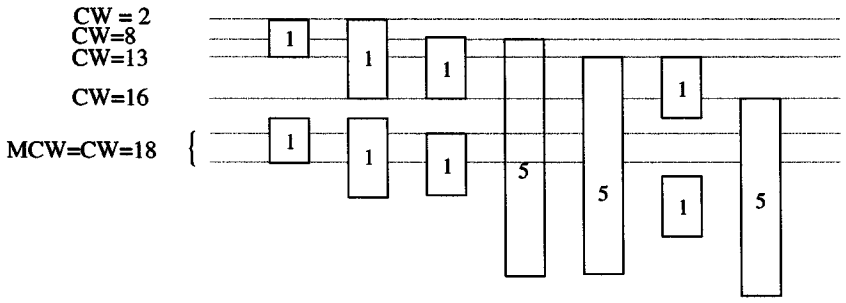

Fig. 20. Example that shows that the MCW can occur at a time that is not the earliest start time of any interval. Numbers in the rectangles denote the width of the intervals.

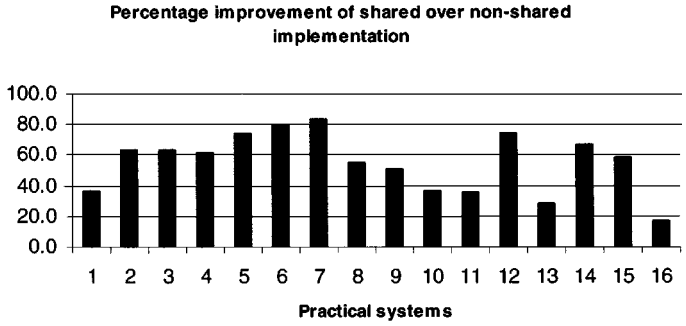

Fig. 21. Improvement percentage of the best shared implementation versus the best nonshared implementation.

ation in sample-rate change ratios. For example, Fig. 22 shows a filter bank with 1/3, 2/3 rate changes; these are the changes that occur across actors c and d for instance. Other ratios that could be used include 1/2, 1/2, or 2/5, 3/5. Similarly, Fig. 23 shows a filter bank 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 filter banks of different depths in Table I. The leftmost column contains the name of the example. The filter bank "qmf23_2d," for example, is a filter bank of depth 2, with 1/3, 2/3 rate changes. Similarly, "qmf235_5d" is a filter bank of depth 5 with rate changes of 3/5 and 2/5. The depth 5, 3, and 2 filter banks have 188, 44, and 20 nodes respectively. "nqmf23_4d" is the one-sided filter bank from Fig. 22. "Satrec" is the satellite receiver example from [29]. The other examples included are "16qamModem," an implementation of a 16-QAM modem; "4pamxmitrec," a transmitter-receiver pair for a 4-PAM signal; "blockVox," an implementation of a vocoder (a system that modulates a synthesized music signal with vocal parameters); "overAddFFT," an implementation of an overlap-add fast Fourier transform (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 [36].

The second column contains the results of running RPMC and postoptimizing with DPPO on these systems, assuming the nonshared model of buffering. This column gives us a 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 dynamic programming heuristic $(sdppo)$ postoptimization for shared buffers on an RPMC generated topological order. 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 topological order). 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 eighth column contains the buffer memory lower bound (BMLB) [4] values for each system. Briefly, the BMLB is a lower bound on the total buffering memory required over all valid SASs, assuming the nonshared model of buffering. The rest of the columns contain the results after applying these heuristics on APGAN-generated topological orders. 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 nonshared implementation; this is computed as shown in the equation at the bottom of the page. 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 filter bank of 1/2, 1/2 rate changes (the most common type of filter bank). It is interesting to note that the methods of Ritz *et al.* [29] 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%.

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. However, most of the best results are on

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

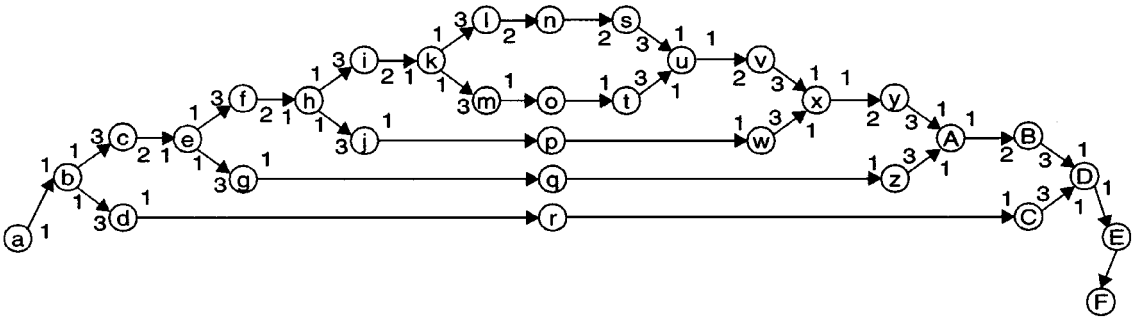

Fig. 22. SDF graph for a one-sided filter bank of depth 4. The produced/consumed numbers not specified are all unity.

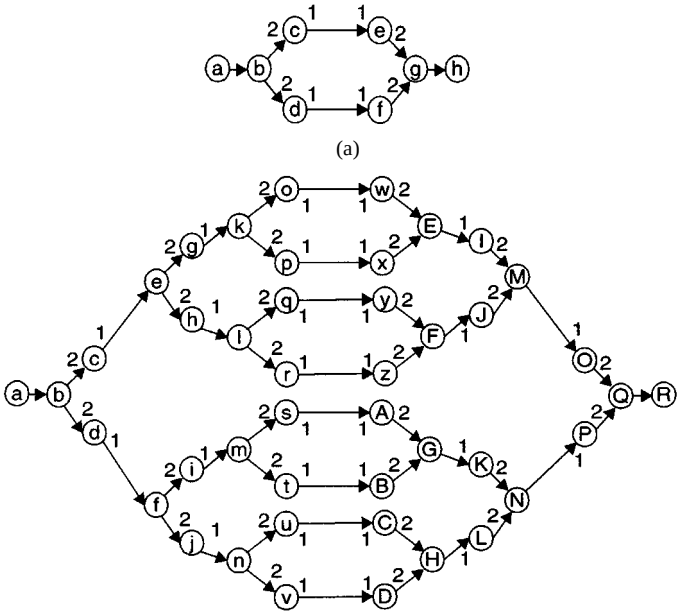

Fig. 23. SDF graph for a two-sided filter bank. (a) Depth 1 filter bank. (b) Depth 3 filter bank. Produced/consumed numbers not specified are all unity.

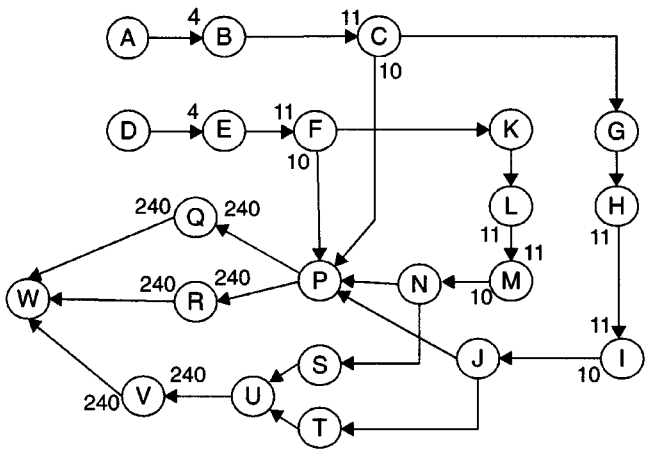

Fig. 24. SDF abstraction for satellite receiver application from [29].

the fairly regular qmf filterbanks; the more irregular systems are apparently better suited to the $APGAN + sdppo + ffdur$ combination.

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 FF 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 the "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 relative improvement over random schedules increases when larger graphs are examined, such as the "qmf12_5d" and "qmd235_5d" examples (these have about 200 nodes each). 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 large amounts of time and compute power to equal or beat the RPMC/APGAN schedules. Hence, we conclude that for compact, shared buffer implementation, APGAN and RPMC are generating topological sorts intelligently, and cannot be easily beaten by nonintelligent strategies such as generating random schedules.

B. 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; 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 Fig. 25. 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

TABLE I
BUFFER SIZES 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
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
overAd dFFT	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

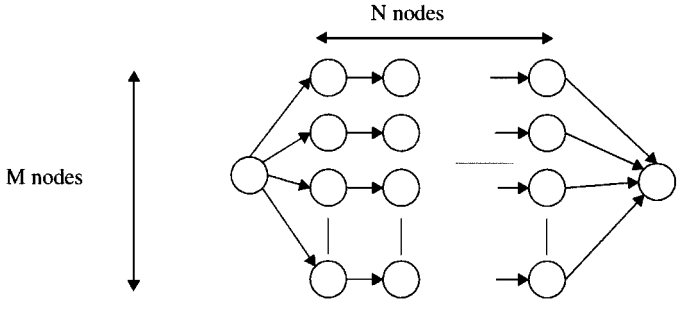

Fig. 25. Homogenous graph for which shared allocation techniques are highly beneficial.

of techniques on this graph for any M and N results in an allocation of $M+1$ units. A nonshared 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.

XI. CONCLUSION

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 dramatically. However, 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 graphical specifications (for instance, a fine-grained description of an finite-impulse response (FIR) filter). Regularity extraction has been applied in the past to high-level synthesis [26], [27] and Kentzer [14] has applied pattern matching algorithms from compiler design to silicon compilers; perhaps these techniques can be applied in the context of SDF compilers. In addition, it would be useful to study techniques that can make use of the regularity implied by the use of hierarchy and graphical higher order functions [18] in dataflow specifications.

ACKNOWLEDGMENT

The authors would like to thank S. Edwards and their anonymous referees, for their helpful remarks for improving the readability and presentation of the paper.

REFERENCES

[1] M. Ade, R. Lauwereins, and J. A. Peperstraete, "Data memory minimization for synchronous data flow graphs emulated on DSP-FPGA targets," in *Proc. Design Automation Conf.*, Anaheim, CA, June 1997, pp. 64–69.

[2] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "Optimal parenthesization of lexical orderings for DSP block diagrams," in *Proc. IEEE Workshop VLSI Signal Processing*, Osaka, Japan, Oct. 1995, pp. 177–186.

[3] S. S. Bhattacharyya and P. K. Murthy, "The CBP parameter—A useful annotation to aid block diagram compilers for DSP," in *Proc. Int. Symp. Circuits and Systems*, Geneva, Switzerland, May 2000, pp. 209–212.

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

[5] S. S. Bhattacharyya, S. Ha, J. Buck, and E. A. Lee, "Generating compact code from dataflow specifications of multirate signal processing algorithms," *IEEE Trans. Circuits Systems-I: Fundamental Theory and Applications*, vol. 42, pp. 138–150, Mar. 1995.

[6] J. T. Buck, S. Ha, D. G. Messerschmitt, and E. A. Lee, "Multirate signal processing in ptolemy," in *Proc. Int. Conf. Acoustics, Speech, and Signal Processing*, Toronto, ON, Canada, Apr. 1991, pp. 1245–1248.

[7] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int. J. Comput. Simulat.*, Jan. 1995.

[8] J. Fabri, *Automatic Storage Optimization*. Ann Arbor, MI: UMI, 1982.

[9] M. R. Garey and D. S. Johnson, *Computers and Intractability—A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1979.

[10] S. Goddard and K. Jeffay, "Managing memory requirements in the synthesis of real-time systems from processing graphs," in *Proc. IEEE Real-Time Technology Applications Symp.*, Denver, CO, June 1998, pp. 59–70.

[11] E. De Greef, F. Catthoor, and H. De Man, "Array placement for storage size reduction in embedded multimedia systems," in *Proc. Int. Conf. Applications Specific Systems, Architectures, Processors*, Zurich, Switzerland, July 1997, pp. 66–75.

[12] J. Horstmannshoff, T. Grotker, and H. Meyr, "Mapping multirate dataflow to complex RT level hardware models," in *Proc. Int. Conf. Application Specific Systems, Architectures, Processors*, Zurich, Switzerland, July 1997, pp. 283–293.

[13] H. A. Kierstead, "Polynomial time approximation algorithm for dynamic storage allocation," *Discrete Math.*, vol. 88, pp. 231–237, 1991.

[14] K. Keutzer, "DAGON: Technology binding and local optimization by DAG matching," in *Proc. Design Automation Conf.*, Miami Beach, FL, June 1987, pp. 617–623.

[15] A. K. Kulkarni, A. Dube, and B. L. Evans, "Benchmarking code generation methodologies for programmable digital signal processors," Dept. Elect. Comput. Eng., Univ. Texas, Austin, Tech. Rep., 1997.

[16] R. Lauwereins, P. Wauters, M. Ade, and J. A. Peperstraete, "Geometric parallelism and cyclo-static data flow in GRAPE-II," in *Proc. IEEE Workshop Rapid System Prototyping*, Grenoble, France, June 1994, pp. 90–107.

[17] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous dataflow programs for digital signal processing," *IEEE Trans. Comput.*, vol. C-36, pp. 24–35, Feb. 1987.

[18] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proc. IEEE*, vol. 83, pp. 773–779, May 1995.

[19] R. Leupers and P. Marwedel, "Retargetable code generation based on structural processor descriptions," *J. Design Automat. Embedded Syst.*, vol. 3, no. 1, pp. 75–108, Jan. 1998.

[20] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang, "Code optimization techniques in embedded DSP microprocessors," *J. Design Automat. Embedded Syst.*, vol. 3, no. 1, pp. 59–73, Jan. 1998.

[21] P. Marwedel and G. Goossens, Eds., *Code Generation for Embedded Processors*. Norwell, MA: Kluwer, 1995.

[22] P. K. Murthy and S. S. Bhattacharyya, "Systematic consolidation of input and output buffers in synchronous dataflow specifications," in *Proc. IEEE Workshop Signal Processing Systems.*, Lafayette, LA, Oct. 2000, pp. 673–682.

[23] ——, "Buffer merging: A powerful technique for reducing memory requirements of synchronous dataflow specifications," Inst. Adv. Comput. Studies, Univ. Maryland, College Park, MD, MIACS-TR-2000-20, Apr. 2000.

[24] P. K. Murthy, S. S. Bhattacharyya, and E. A. Lee, "Joint minimization of code and data for synchronous dataflow programs," *J. Formal Methods Syst. Design*, vol. 11, no. 1, pp. 41–70, July 1997.

[25] P. K. Murthy and S. S. Bhattacharyya, "Approximation algorithms and heuristics for the dynamic storage allocation problem," Univ. Maryland Inst. Adv. Comput. Studies, College Park, MD, http://www.cs.umd.edu/TRs/TRumiacs.html, May 1999.

[26] D. S. Rao and F. J. Kurdahi, "An approach to scheduling and allocation using regularity extraction," in *Proc. Eur. Conf. Design Automation*, Paris, France, Feb. 1993, pp. 557–561.

[27] ——, "On clustering for maximal regularity extraction," *IEEE Trans. Comput.-Aided Design Integrated Circuits Syst.*, vol. 12, pp. 1198–1208, Aug. 1993.

[28] S. Ritz, M. Pankert, and H. Meyr, "Optimum vectorization of scalable synchronous dataflow graphs," in *Proc. Int. Conf. Applications Specific Array Processors*, Venice, Italy, Oct. 1993, pp. 285–296.

[29] S. Ritz, M. Willems, and H. Meyr, "Scheduling for optimum data memory compaction in block diagram oriented software synthesis," in *Proc. ICASSP*, Detroit, MI, May 1995, pp. 2651–2654.

[30] R. Sethi, "Complete register allocation problems," *SIAM J. Computing*, vol. 4, no. 3, pp. 226–248, Sept. 1975.

[31] W. Sung, J. Kim, and S. Ha, "Memory efficient synthesis from dataflow graphs," in *Int. Symp. System Synthesis*, Hinschu, Taiwan, Dec. 1998, pp. 137–142.

[32] P. P. Vaidyanathan, *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice-Hall, 1993.

[33] J. Vanhoof, I. Bolsens, and H. De Man, "Compiling multi-dimensional data streams into distributed DSP ASIC memory," in *Proc. Int. Conf. Computer-Aided Design*, Santa Clara, CA, Nov. 1991, pp. 272–275.

[34] M. C. Williamson, "Synthesis of parallel hardware implementations from synchronous dataflow graph specifications," Ph.D. dissertation, Electron. Res. Lab., Univ. California, Berkeley, CA, June 1998.

[35] V. Zivojinovic, J. M. Velarde, C. Schlager, and H. Meyr, "DSPStone—A DSP-oriented Benchmarking methodology," in *Int. Conf. Signal Processing Application Technology*, Dallas, TX, Oct. 1994, pp. 715–720.

[36] "The ptolemy almagest,", http://ptolemy.eecs.berkeley.edu.

[37] P. K. Murthy and S. S. Bhattacharyya, "Buffer merging: A powerful technique for reducing memory requirements of synchronous dataflow specifications," in *Proc. Int. Symp. System Synthesis*, San Jose, CA, Nov. 1999, pp. 78–84.

Praveen K. Murthy (S'88–M'96) received his B.S.E.E. degree from the Georgia Institute of Technology, Atlanta, in 1989, and the M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California at Berkeley in 1993 and 1996, respectively.

He is at Angeles Design Systems, San Jose, CA, where he holds the position of Distinguished Member of Technical Staff. Prior to joining Angeles, he was at Cadence Design Systems as a member of Consulting Staff. He has consulted for BDTI, Berkeley, CA, in the area of digital signal processor (DSP) architectures. He is a coauthor of *Software Synthesis from Dataflow Graphs* (Norwell, MA: Kluwer, 1996). His research interests span all areas of system level design and synthesis including simulation, techniques for producing optimized software implementations, semantics of different models of computation, multidimensional dataflow, and software tools for rapid prototyping.

Shuvra S. Bhattacharyya (S'87–M'95) received the B.S. degree from the University of Wisconsin at Madison, and the Ph.D. degree from the University of California at Berkeley, 1987 and 1994, respectively.

He is an Assistant Professor in the Department of Electrical and Computer Engineering and the Institute for Advanced Computer Studies, University of Maryland, College Park. He has held industrial positions as a Researcher at Hitachi, San Jose, CA, and as a Compiler Developer at Kuck & Associates, Urbana, IL. He has consulted for industry in the areas of compiler techniques and multiprocessor architectures for embedded systems. He is the coauthor of two books and the author or coauthor of more than 30 refereed technical articles. His research interests center around architectures and computer-aided design for embedded systems.

Dr. Bhattacharyya is a recipient of the NSF Career Award.