Technical Report #UMIACS-TR-2000-20, Institute for Advanced Computer Studies,
University of Maryland at College Park, April, 2000.

Buffer Merging — A Powerful Technique for Reducing Memory Requirements of Synchronous Dataflow Specifications

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

Shuvra S. Bhattacharyya
University of Maryland, College Park
ssb@eng.umd.edu

Abstract[1]

In this paper, we develop a new technique called buffer merging for reducing memory requirements of synchronous dataflow (SDF) specifications. SDF has proven to be an attractive model for specifying DSP systems, and is used in many commercial tools like DSPCanvas, SPW, and COSSAP. Good synthesis from an SDF specification depends crucially on scheduling, and memory is an important metric for generating efficient schedules. Previous techniques on memory minimization have either not considered buffer sharing at all, or have done so at a fairly coarse level (the meaning of this will be made more precise in the paper). In this paper, we develop a buffer overlaying strategy that works at the level of an input/output edge pair of an actor. It works by algebraically encapsulating the lifetimes of the tokens on the input/output edge pair, and determines the maximum amount of the input buffer space that can be reused by the output. We develop the mathematical basis for performing merging operations, and develop several algorithms and heuristics for using the merging technique for generating efficient implementations. We show improvements of up to 54% over previous techniques.

1 Introduction

Memory is an important metric for generating efficient code for DSPs used in embedded applications. This is because most DSPs have very limited amounts of on-chip memory, and adding off-chip memory is frequently not a viable option due to the speed, power, and cost penalty this entails. High-level language compilers, like C compilers have been ineffective for generating good DSP code [20]; this is why most DSPs are still programmed manually in assembly language. However, this is a tedious, error-prone task at best, and the increasing complexity of the systems being implemented, with shorter design cycles, will require design development from a higher level of abstraction.

1. A portion of this research was sponsored by the US National Science Foundation (Project MIP9734275).

One potential approach is to do software synthesis from block-diagram languages. Block diagram environments for DSPs have proliferated recently, with industrial tools like DSPCanvas from Angeles Design Systems, and COSSAP [16] from Synopsys, and academic tools like Ptolemy [4] from UC Berkeley, and GRAPE from K. U. Leuven [6]. Reasons for their popularity include ease-of-use, intuitive semantics, modularity, and strong formal properties of the underlying dataflow models.

Most block diagram environments for DSPs that allow software synthesis, use the technique of threading for constructing software implementations. In this method, the block diagram is scheduled first. Then the code-generator steps through the schedule, and pieces together code for each actor that appears in the schedule by taking it from a predefined library. The code generator also performs memory allocation, and expands the macros for memory references in the generated code.

Clearly, the quality of the code will be heavily dependent on the schedule used. Hence, 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 general purpose compiler cannot make usually use of the global control and dataflow that our scheduler can exploit. Thus, the techniques we develop in this paper are complementary to the work being done on developing better procedural language compilers for DSPs [8][9]. Since the individual actors are programmed in procedural languages like ‘C’, the output of our SDF compiler is sent to a procedural language compiler to optimize the internals of each actor, and to possibly further optimize the code at a global level (for example, by performing global register allocation.) 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.

The specific problem addressed by this paper is the following. Given a schedule for an SDF graph, there are several strategies that can be used for implementing the buffers needed on the edges of the graph. Previous work on minimizing these buffer sizes has used two models: implementing each buffer separately (for example, in [1][2][18]), or using lifetime analysis techniques for sharing buffers (for example, in [5][14][17]). In this paper, we present a third strategy—buffer merging. This strategy allows sharing of input and output buffers systematically, something that the lifetime-based approaches of [14][17] are unable to do. The reason that lifetime-based approaches break down when input/output edges are considered is because they make the conservative assumption that an output buffer becomes live as soon as an actor begins firing, and that an input buffer does not die until the actor has finished execution. Hence, the lifetimes of the input and output buffers overlap, and they cannot be shared. However, as we will show in this paper, relaxing this assumption by analyzing the production and consumption pattern of individual tokens results in significant reuse opportunities that can be efficiently exploited. However, the merging approach of this paper is complementary to lifetime-based approaches because the merging technique is not able to exploit global sharing opportunities based on the topology of the graph and the schedule. It can only exploit sharing opportunities at the input/output level, based on a fine-grained analysis of token traffic during a single, atomic execution of an actor. Thus, we give a hybrid algorithm that combines both of these techniques and show that dramatic reductions in memory usage are possible compared to either technique used by itself.

In a synthesis tool called ATOMIUM, De Greef, Catthoor, and De Man have developed lifetime analysis and memory allocation techniques for single-assignment, static control-flow specifications that involve explicit looping constructs, such as for loops [5]. While the techniques in [5] are able to reuse and overlay variables and arrays very effectively, the worst case complexity of the algorithms used is exponential. Our algorithms used in this paper, in contrast, provably run in polynomial-time because we are able to exploit the particular, restricted structure of SDF programs and single-appearance schedules for these programs. The algorithms we develop are also purely graph-theoretic techniques, and do not use ILP formulations or array subscript analysis, problems that can have prohibitive complexity in general.

The CBP parameter that we develop in section 5.1, and more completely in [3], plays a role that is somewhat similar to the array index distances derived in the in-place memory management strategies of Cathedral [19], which applies to nested loop constructs in Silage. The merging approach presented in this paper is different from the approach of [19] in that it is specifically targeted to the high regularity and modularity present in single appearance schedule implementations (at the expense of decreased generality). In particular, the CBP-based overlapping of SDF input/output buffers by shifting actor read and write pointers does not emerge in any straightforward way from the more general techniques developed in [19]. Our form of buffer merging is especially well-suited for incorporation with the SDF vectorization techniques (for minimizing context-switch overhead) developed at the Aachen University of Technology [16] since the absence of nested loops in the vectorized schedules allows for more flexible merging of input/output buffers.

Ritz et al. [17] give an enumerative method for reducing buffer memory in SDF graphs. Their approach operates only on flat single appearance schedules since buffer memory reduction is tertiary to their goal of reducing code size and context-switch overhead (for which flat schedules are better). However, it's been shown in [2] that on practical applications, their method yields buffering requirements that can be much larger than using nested schedules with each buffer implemented separately.

In [18], Sung et al. explore an optimization technique that combines procedure calls with inline code for single appearance schedules; this is beneficial whenever the graph has many different instantiations of the same basic actor. Thus, using parametrized procedure calls enables efficient code sharing and reduces code size even further. Clearly, all of the scheduling techniques mentioned in this paper can use this code-sharing technique also, and our work is complementary to this optimization.

2 Notation and background

Dataflow is a natural model of computation to use as the underlying model for a block-diagram language for designing DSP systems. The blocks in the language correspond to actors in a dataflow graph, and the connections correspond to directed edges between the actors. These edges not only represent communication channels, conceptually implemented as FIFO queues, but also establish precedence constraints. An actor fires in a dataflow graph by removing tokens from its input edges and producing tokens on its output edges. The stream of tokens produced this way corresponds naturally to a discrete time signal in a DSP system. In this paper, we consider a subset of dataflow called synchronous dataflow (SDF) [7]. 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.

Fig. 1(a) shows a simple SDF graph. Each edge is annotated with the number of tokens produced (consumed) by its source (sink) actor. Given an SDF edge e, we denote the source actor, sink actor, and delay (initial tokens) of e by $src(e)$, $snk(e)$, and $del(e)$. Also, $prd(e)$ and $cns(e)$ denote the number of tokens produced onto e by $src(e)$ and consumed from e by $snk(e)$. If $prd(e) = cns(e) = 1$ for all edges e, the graph is called **homogenous**. In general, each edge has a FIFO buffer; the number of tokens in this buffer defines the state of the edge. Initial tokens on an edge are just initial tokens in the buffer. The size of this buffer can be determined at compile time, as shown below. The state of the graph is defined by the states of all edges.

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 (i.e, returns the graph to its initial state). We represent the minimum number of times each actor must be fired in a valid schedule by a vector q_G, indexed by the actors in G (we often suppress the subscript if G is understood). These minimum numbers of firings can be derived by finding the minimum positive integer solution to the **balance equations** for G, which specify that q must satisfy $prd(e)q(src(e)) = cns(e)q(snk(e))$, for all edges e in G.

The vector q, when it exists, is called the **repetitions vector** of G, and can be computed efficiently [2].

3 Constructing memory-efficient loop structures

In [2], the concept and motivation behind **single appearance schedules** (**SAS**) has been defined and shown to yield an optimally compact inline implementation of an SDF graph with regard to code size (neglecting the code size overhead associated with the loop control). An SAS is one where each actor appears only once when loop notation is used. Figure 1 shows an SDF graph, and valid schedules for it. The notation $2B$ represents the firing sequence BB. Similarly, $2(B(2C))$ represents the schedule loop with firing sequence $BCCBCC$. We say that the **iteration count** of this loop is 2, and the body of this loop is $B(2C)$. Schedules 2 and 3 in figure 1 are single appearance schedules since actors A, B, C appear only once. An SAS like the third one in Figure 1(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.

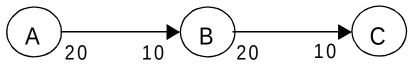
Scheduling can also have a significant impact on the amount of memory required to implement the buffers on the edges in an SDF graph. For example, in Figure 1(b), the buffering requirements for the four schedules, assuming that one separate buffer is implemented for each edge, are 50, 40, 60, and 50 respectively.

4 Optimizing for buffer memory

We give priority to code-size minimization over buffer memory minimization; the importance of addressing this prioritization is explained in [2][12]. Hence, the problem we tackle is one of finding buffer-memory-optimal SAS, since this will give us the best schedule in terms of buffer-memory consumption amongst the schedules that have minimum code size. Following [2] and [12], we also concentrate on acyclic SDF graphs since algorithms for acyclic graphs can be used in the general SAS framework developed in [2].

For an acyclic SDF graph, any topological sort $a\ b\ c\ldots$ immediately leads to a valid flat SAS given by $(q(a)a)\ (q(b)b)\ldots$. Each such flat SAS leads to a set of SASs corresponding to different nesting orders.

(a)



(b)

Valid Schedules

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

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

Fig 1. An example used to illustrate the interaction between scheduling SDF graphs and the memory requirements of the generated code.

In [12] and [2], we define the buffering cost as the sum of the buffer sizes on each edge, assuming that each buffer is implemented separately, without any sharing. With this cost function, we give a post-processing algorithm called dynamic programming post optimization (**DPPO**) that organizes a buffer-optimal nested looped schedule for any given flat SAS. We also develop two heuristics for generating good topological orderings, called APGAN and RPMC.

In this paper, we use an alternative cost for implementing buffers. Our cost is based on overlaying buffers so that spaces can be re-used when the data is no longer needed. This technique is called **buffer merging**, since, as we will show, merging an input buffer with an output buffer will result in significantly less space required than their sums.

5 Merging an input/output buffer pair

Example 1: Consider the second schedule in figure 1(b). If each buffer is implemented separately for this schedule, the required buffers on edges AB and BC will be of sizes 20 and 20, giving a total requirement of 40. Suppose, however, that it is known that B consumes its 10 tokens per firing *before* it writes any of the 20 tokens. Then, when B fires for the first time, it will read 10 tokens from the buffer on AB, leaving 10 tokens there. Now it will write 20 tokens. At this point, there are 30 live tokens. If we continue observing the token traffic as this schedule evolves, it will be seen that 30 is the maximum number that are live at any given time. Hence, we see that in reality, we only need a buffer of size 30 to implement AB and BC. Indeed, the diagram shown in figure 2 shows how the read and write pointers for actor B would be overlaid, with the pointers moving right as tokens are read and written. As can be seen, the write pointer, X(w,BC) never overtakes the read pointer X(r,AB), and the size of 30 suffices. Hence, we have merged the input buffer (of size 20) with the output buffer (of size 20) by overlapping a certain amount that is not needed because of the lifetimes of the tokens.

In order to merge buffers in this manner systematically, we introduce several new concepts, notation, and theorems. We assume for the rest of the paper that our SDF graphs are delayless because initial tokens on edges may have lifetimes much greater than tokens that are produced and consumed during the schedule, thus rendering the merging incorrect. This is not a big restriction, since if there are delays on edges, we can divide the graph into regions that are delayless, apply the merging techniques in those por-

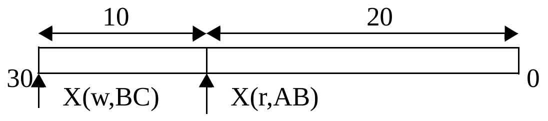


Fig 2. The merged buffer for implementing edges AB and BC in figure 1.

tions, and allocate the edges with delays separately. In practical systems, the number of edges having initial tokens is usually a small percentage of the total number of edges, especially in acyclic SDF systems or subsystems. We could even use retiming techniques to move delays around and try to concentrate them on a few edges so that the delayless region becomes as big as possible. Retiming to concentrate delays in this manner has been studied in a different context in [21]. The objective there is to facilitate more extensive vectorization of the input SDF graph.

5.1 The CBP parameter

We define a parameter called the **consumed-before-produced (CBP)** value; this parameter is a property of the SDF actor and a particular input/output edge pair of that actor. Informally, it gives the best known lower bound on the difference between the number of tokens consumed and number of tokens produced over the entire time that the actor is in the process of firing. Formally, let X be an SDF actor, let e_i be an input edge of X, and e_o be an output edge of X. Let the firing of X begin at time 0 and end at time T. Define $c(t)$ $(p(t))$ to be the number of tokens that have been consumed (produced) from (on) e_i (e_o) by time $t \in [0, T]$. The quantities $c(t)$ and $p(t)$ are monotonically nondecreasing functions that increase from 0 to $cns(e_i)$ and $prd(e_o)$ respectively. Then, we define

$$CBP(X, e_i, e_o) = MIN_t\{c(t) - p(t)\}. \qquad \textbf{(EQ 1)}$$

Note that at $t = 0$, nothing has been consumed or produced, so $c(0) - p(0) = 0$. At T, $p = prd(e_o)$ tokens have been produced and $c = cns(e_i)$ tokens have been consumed; hence, $c(T) - p(T) = c - p$. So, we immediately have

$$-p \le CBP(X, e_i, e_o) \le MIN(0, c - p). \qquad \textbf{(EQ 2)}$$

There are several ways in which the CBP parameter could be determined. The simplest would be for the programmer of the actor to state it based on analyzing the written code inside the actor. This analysis is quite simple in many cases that occur commonly; a study of this type of analysis is reported in [3]. Automatic deduction by source code analysis could also be done, but is beyond the scope of this paper. An

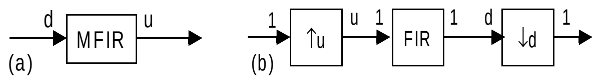


Fig 3. Polyphase FIR filter. The structure in (a) implements the graph (b) efficiently.

analysis of optimized assembly language code written for the polyphase FIR filter in the Ptolemy library (figure 3) shows that [3]

$$CBP(MFIR) = \begin{cases} 0 \text{ if } (u \le d) \\ (d-u) \text{ if } (u > d) \end{cases}. \qquad \textbf{(EQ 3)}$$

Such filters are commonly used in DSP and communication systems. For small, homogenous actors like adders and multipliers, $CBP = 0$. If it is not possible to determine a good (meaning largest) lower bound for the CBP parameter, then the worst-case bound of $-p$ is assumed. As we will show, better bounds for CBP will enable smaller merged buffers.

5.2 R-Schedules and the Schedule Tree

As shown in [12], it is always possible to represent any single appearance schedule for an acyclic graph as

$$(i_L S_L)(i_R S_R), \qquad \textbf{(EQ 4)}$$

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 equation 4. SASs having this form at all levels of the loop hierarchy are called R-schedules [12].

Given an R-schedule, we can represent it naturally as a binary tree. The internal nodes of this tree will contain the iteration count of the subschedule rooted at that node. Subschedules S_L or S_R that only have one actor become leaf nodes containing the iteration count and the actor. Figure 4 shows schedule trees for the SAS in figure 1. Note that a schedule tree is not unique since if there are loop factors of 1, then the split into left and right subgraphs can be made at multiple places. In figure 4, the schedule tree for the

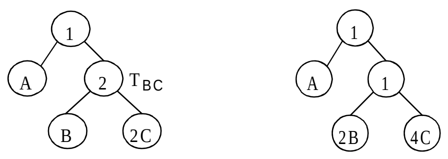


Fig 4. Schedule trees for schedules in figure 1(b)(2) and (3)

flat SAS in figure 1(b)(3) is based on the split $\{A\}\{B, C\}$. However, we could also take the split to be $\{A, B\}\{C\}$. As we will show below, the cost function will not be sensitive to which split is used as they both represent the same schedule.

Define $lf(v)$ to be the iteration count of the node v in the schedule 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. A subtree S is a **subset** of a subtree T, $S \subseteq T$ if there is a node v in T such that $S = subtree(v)$. A subtree S is a **strict subset** of a subtree T, $S \subset T$ if there is a node $v \neq root(T)$ in T such that $S = subtree(v)$.

Consider a pair of input/output edges e_i, e_o for an actor Y. Let $X = src(e_i)$, $Z = snk(e_o)$ $snk(e_i) = Y = src(e_o)$. Let T_{XYZ} be the smallest subtree of the schedule tree that contains the actors X, Y, Z. Similarly, let $T_{XYZ'}$ be the largest subtree of T_{XYZ} containing actors X, Y, but not containing Z. In figure 4, T_{ABC} is the entire tree, and $T_{A'BC}$ is the tree rooted at the node marked T_{BC}. Largest simply means the following: for every tree $T' \subseteq T_{XYZ}$ that contains X, Y and not Z, $T_{XYZ'} \supseteq T'$. Smallest is defined similarly. Let G be an SDF graph, S be an SAS, and $T(G, S)$ be the schedule tree representing S.

Definition 1: The edge pair $\{e_i, e_o\}$ is said to be **output dominant (OD)** with respect to $T(G, S)$ if $T_{XYZ'} \subset T_{XYZ}$ (note that $\subset$ denotes the strict subset).

Definition 2: The edge pair $\{e_i, e_o\}$ is said to be **input dominant (ID)** with respect to $T(G, S)$ if $T_{X'YZ} \subset T_{XYZ}$.

The edge pair $\{AB, BC\}$ is ID with respect to both the schedule trees depicted in figure 4. Intuitively, an OD edge pair results from X, Y being more deeply nested together in the SAS than Z.

Fact 1: For any edge pair $\{e_i, e_o\}$, and actors X, Y, Z as defined above, $\{e_i, e_o\}$ is either OD or ID with respect to $T(G, S)$.

Definition 3: For an OD edge pair $\{e_i, e_o\}$, and actor $snk(e_i) = Y = src(e_o)$, let I_1 be the product of the loop factors in all nodes on the path from the leaf node containing Y to the root node of $T_{XYZ'}$. I_1 is simply the total number of invocations of Y in the largest subschedule not containing Z. Similarly, let I_2 be the product of all the loop factors on the path from the leaf node containing Y to the root node of the subtree $T_{X'Y} \subseteq T_{XYZ'}$, where $T_{X'Y}$ is taken to be the largest tree containing Y but not X.

Definition 4: For an ID edge pair $\{e_i, e_o\}$, and actor $snk(e_i) = Y = src(e_o)$, let I_1 be the product of the loop factors in all nodes on the path from the leaf node containing Y to the root node of $T_{X'YZ}$. I_1 is simply the total number of invocations of Y in the largest subschedule not containing X. Similarly, let I_2 be

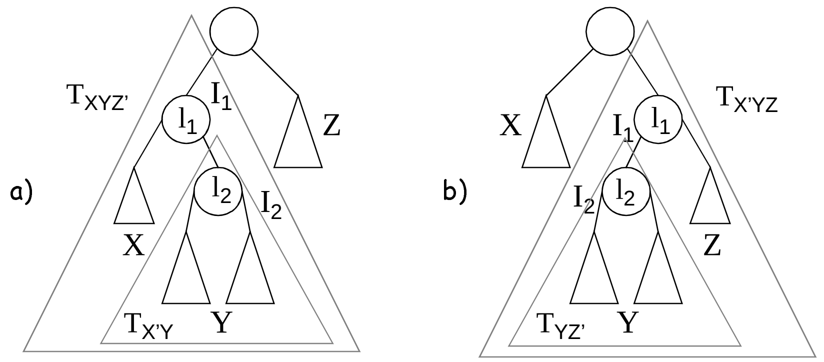


Fig 5. a) The schedule tree for an OD edge pair. b) Schedule tree for an ID edge pair.

the product of all the loop factors on the path from the leaf node containing Y to the root node of the sub-tree $T_{YZ'} \subseteq T_{X'YZ}$, where $T_{YZ'}$ is taken to be the largest tree containing Y but not Z.

Note that $I_1 = lf(root(T))I_2$ where $T = T_{XYZ'}$ for OD edges pairs and $T = T_{X'YZ}$ for ID edge pairs.

In figure 4, for the schedule tree on the left, we have $I_1 = 2, I_2 = 1$ for B, and $I_1 = 2, I_2 = 2$ for B in the tree on the right.

5.3 Buffer merging formulae

5.3.1 Merging an input/output buffer pair

Given these definitions, we can prove the following theorem about the size of the merged buffer.

Theorem 1: Let an input-output edge pair $\{e_i, e_o\}$ of an actor Y, and a SAS S for the SDF graph G be given. Define $p = prd(e_o)$ and $c = cns(e_i)$. The total size of the buffer required to implement this edge pair is given by the following table:

Table 1: Size of the merged buffer

	OD	**ID**
$c - p < 0$	$I_1 p + c - p + \lvert CBP \rvert$	$I_1 c + I_2(p - c) + c - p + \lvert CBP \rvert$
$c - p \geq 0$	$I_1 p + I_2(c - p) + \lvert CBP \rvert$	$I_1 c + \lvert CBP \rvert$

Proof: Let $src(e_i) = X$ and $snk(e_o) = Z$.

Case 1: ($c - p < 0$, OD):

The schedule tree for this case will be as shown in figure 5(a). The triangles in the figure represent sub-trees. Notice that $I_1 = l_1 I_2$, where $l_1 \geq 1$. On edge e_i, $I_2 c$ tokens are transferred during the execution of the loop represented by the subtree $T_{X'Y}$, and on edge e_o, $I_1 p$ tokens are produced during the execution of the schedule represented by $T_{XYZ'}$. Since $I_1 p > I_2 c$, consider the overlaid buffer shown in figure 6. We need to determine what N should be so that the write pointer X_{wp} on edge e_o never overtakes the read pointer X_{rp} on edge e_i. Now, as the schedule represented by the schedule subtree $T_{XYZ'}$ is executed, there will be l_1 executions of the schedule subtree $T_{X'Y}$. Hence, after I_2 executions of Y, the read pointer will come back to the position shown in figure 6, and will move right as $T_{X'Y}$ is executed. After $0 \le k \le l_1 - 1$ executions of $T_{X'Y}$, there will have been $kI_2 p$ tokens produced on edge e_o, and there will be another $I_2 c$ tokens that need to be consumed during the next execution of $T_{X'Y}$. Hence, there are $kI_2 p + I_2 c$ live tokens. Clearly, this will reach the maximum at $k = l_1 - 1$; then, there will be $(l_1 - 1)I_2 p + I_2 c = I_1 p + I_2(c - p) < I_1 p$ live tokens since $c - p < 0$. Note that after l_1 executions of $T_{X'Y}$, there will be $I_1 p$ live tokens since this is the last execution of $T_{X'Y}$, and there is no need to consume $I_2 c$ tokens again. Hence, let $N = I_1 p$. Now, after k iterations of $T_{X'Y}$, we need to verify whether $X_{wp} = N - kI_2 p \ge X_{rp} = I_2 c$. Clearly, the inequality holds since at $k = l_1 - 1$, we have $X_{wp} = I_2 p$. Now consider the very last invocation of Y. At this point, $X_{wp} = I_2 p - (I_2 - 1)p = p$, and $X_{rp} = c$. During this last invocation, c tokens will be consumed and p will be produced. If Y produces the p tokens before it consumes any of the c tokens (meaning that $CBP = -p$), then the c tokens will be overwritten. Hence, we need to augment the entire buffer by an amount x such that the following is satisfied: before the very last execution of Y, $X_{wp} = x + p$, and $X_{rp} = c$. Let the last invocation of Y take time T, and consider the state of the buffer after time $0 \le t \le T$: $X_{wp} = x + p - p(t)$ and $X_{rp} = c - c(t)$, where $p(t)$ and $c(t)$ are the number of tokens produced and consumed by X by time t. We want $X_{wp} \ge X_{rp}$; hence,

$$x \ge c - p + p(t) - c(t) . \qquad \textbf{(EQ 5)}$$

Since $c(t) - p(t) \ge CBP$, the right hand side of equation 5 is maximized when $p(t) - c(t) = -CBP$. So, $x \ge c - p - CBP = c - p + |CBP|$ as $CBP \le 0$. Therefore, the least amount of augmentation required is $c - p + |CBP|$ and $N = I_1 p + c - p + |CBP|$.

Case 2: $c - p \ge 0$**, OD.**

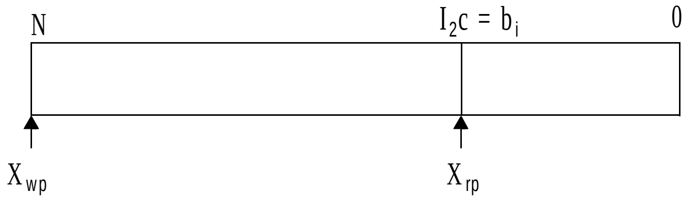


Fig 6. An overlaid buffer for the OD cases.

Using the same analysis as for case 1, we have that the maximum number of live tokens is reached when $k = l_1 - 1$ in the expression $kI_2p + I_2c$. In other words,

$$kI_2p + I_2c = (l_1 - 1)I_2p + I_2c = I_1p + I_2(c - p) \geq I_1p .$$

Now we need to verify whether X_{wp} will overtake X_{rp}. After k invocations of $T_{X'Y}$, $X_{wp} = N - kI_2p = I_1p + I_2(c-p) - kI_2p$, and $X_{rp} = I_2c$. Since we want $X_{wp} \geq X_{rp}$, we get that $I_1 - I_2 \geq kI_2$, or $k \leq l_1 - 1$ as required. At $k = l_1 - 1$, $X_{wp} = X_{rp} = I_2c$ and the effect of the CBP will need to be taken into account now. Again, if all c tokens are consumed before any p are produced, X_{wp} will never overtake X_{rp} since $c - p \geq 0$. However, if $CBP < 0$, then this may not be true. So we need to determine the amount x by which X_{wp} should be moved to the right when $k = l_1 - 1$ so that X_{wp} will never overtake X_{rp}. Consider the first invocation of Y during the last (i.e, l_1 th) invocation of $T_{X'Y}$. Again, letting this invocation of Y take time T, consider the state of the buffer after time $0 \leq t \leq T$: $X_{wp} = x + I_2c - p(t)$ and $X_{rp} = I_2c - c(t)$. Since we want $X_{wp} \geq X_{rp}$, we have $x \geq p(t) - c(t)$, and $p(t) - c(t) \leq -CBP$. Hence, we have $x \geq -CBP = |CBP|$. If we make $x = |CBP|$, we have that for the ith iteration of Y during l_1th invocation of $T_{X'Y}$, $X_{wp} = |CBP| + I_2c - ip$ and $X_{rp} = I_2c - ic$, and clearly, $|CBP| \geq i(p-c)$ since $p - c < 0$. In summary, for the second case, the size of the overlaid buffer is $N = I_1p + I_2(c-p) + |CBP|$.

Case 3: $c - p < 0$, ID.

The schedule tree for this case will be as shown in figure 5(b). On edge e_i, I_1c tokens are transferred during the execution of the loop represented by the subtree $T_{X'YZ}$, and on edge e_o, I_2p tokens are produced during the execution of the schedule represented by $T_{YZ'}$. Consider the overlaying strategy shown in figure 7. Here, the read pointer X_{rp} reads the I_1c tokens between locations I_1c to zero as shown. The write pointer X_{wp} advances from N, where N has to be determined, to I_1c, and wraps around back to N since actor Z will then read these tokens written by Y. Clearly, $N = I_2p + I_1c$ is feasible; no overlaying occurs in this case. However, we can determine a smaller value of N that permits reuse of the space vacated by the tokens consumed from location I_1c onwards (towards 0). Since the I_1c tokens on e_i are steadily consumed during the l_1 invocations of $T_{YZ'}$, and since the I_2p tokens produced on e_o are

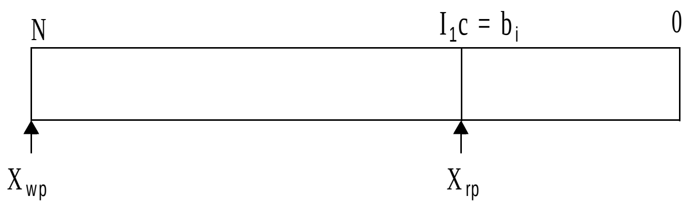


Fig 7. An overlaid buffer for the ID cases.

reused, there are only two points to consider for maximum number of live tokens: before any executions of $T_{YZ'}$ have taken place, when there are I_1c live tokens, and after one execution of $T_{YZ'}$, when there are $I_1c - I_2c + I_2p = I_1c + I_2(p - c)$ live tokens. Since $c < p$, the maximum number of live tokens is $N = I_1c + I_2(p - c)$. To verify whether $X_{wp} \geq X_{rp}$, we have after i invocations of $T_{YZ'}$, that $X_{wp} = I_1c + I_2(p - c) - ip$, and $X_{rp} = I_1c - ic$. Since $i \leq I_2$ and $c < p$, $X_{wp} \geq X_{rp}$ is satisfied. At $i = I_2$, $X_{wp} = X_{rp}$, but now, the next execution of Y will result in X_{wp} wrapping back to N, while X_{rp} will remain where it is. However, at $i = I_2 - 1$, $X_{wp} = I_1c - I_2c + p$, and $X_{rp} = I_1c - I_2c + c$. If the $CBP = -p$, then all p tokens will be produced before any c have been consumed, meaning that $X_{wp} < X_{rp}$ during the I_2th invocation of Y in the first invocation of $T_{YZ'}$. In order to account for the CBP, we again have to determine the x by which X_{wp} should be moved to the left so that X_{wp} will never overtake X_{rp}. Using the same notation as before, we have $X_{wp} = x + I_1c - I_2c + p - p(t)$ and $X_{rp} = I_1c - I_2c + c - c(t)$ after time $0 \leq t \leq T$ during the I_2th invocation of Y in the first invocation of $T_{YZ'}$. Requiring $X_{wp} \geq X_{rp}$ implies that $x \geq p(t) - c(t) + c - p$. Since $c(t) - p(t) \geq CBP$, we have that $p(t) - c(t) + c - p \leq -CBP + c - p = c - p + |CBP|$; hence, $x = c - p + |CBP|$ is the least feasible increment. In summary, $N = I_1c + I_2(p - c) + c - p + |CBP|$ for case 3.

Case 4: $c - p \geq 0$**, ID.**

In this case, $I_1c + I_2(p - c) \leq I_1c$, and hence the maximum number of live tokens occurs before any invocations of Y in $T_{YZ'}$ have taken place at all. If $N = I_1c$, then $X_{wp} = X_{rp}$, and we need to again account for the CBP by determining the amount x by which we should increment N. Using the same analysis techniques, we need that $x + I_1c - p(t) \geq I_1c - c(t)$, or $x \geq p(t) - c(t)$. This means that the least value of x that satisfies the requirement is $x = |CBP|$; hence, $N = I_1c + |CBP|$. Note that since $c \geq p$, the read pointer will move left faster than the write pointer, so $X_{wp} \geq X_{rp}$ will hold for all subsequent invocations of Y. **QED**.

If the edge pair can be regarded as either OD or ID (this happens if $I_1 = I_2$), then the expressions in the 3rd column equal those in the 2nd column. Similarly, if $c = p$, then the expressions in the second row coincide with the expressions in the 3rd row. This verifies our assertion that it does not matter where the split is taken in the SAS when there are multiple choices. Note also that better lower bounds for the CBP make it less negative, reducing $|CBP|$, and thus the size of the merged buffer.

Lemma 1: The size of the merged buffer is no greater than the sum of the buffer sizes implemented separately.

Proof: The sum of the buffer sizes when implemented separately is given by $I_1p + I_2c$ and $I_1c + I_2p$ for the OD and ID cases. For case 1 (as defined in the proof of theorem 1), we need to verify whether $I_2c \geq c - p + |CBP|$. Since $c - p \leq 0$, we have $p - c \leq |CBP| \leq p$. Hence, $0 \leq c - p + |CBP| \leq c$, and $I_2c \geq c$. For case 2, we need to verify whether $I_2c \geq I_2(c-p) + |CBP|$, or $I_2p \geq |CBP|$, which clearly holds due to equation 2. For case 3, we need to verify that $I_2p \geq I_2(p-c) + c - p + |CBP|$, or $I_2c \geq c - p + |CBP|$, as before. For case 4, the inequality to be satisfied is $I_2p \geq |CBP|$, as before.

Observation 1: For the MFIR of fig. 3, table 1 becomes

Table 2: Merged buffer size for MFIR

	OD	ID
$c - p < 0$	I_1p	$I_1c + I_2(p-c)$
$c - p \geq 0$	$I_1p + I_2(c-p)$	I_1c

Observation 2: For an input-output edge pair $\{e_i, e_o\}$ of an actor Y with $cns(e_i) = c = p = prd(e_o)$, and $CBP = 0$, table 1 simplifies to $I_1p = I_1c$ for all cases.

Homogenous actors are a common special case where observation 2 holds. In the remainder of the paper, we will assume that for illustrative examples, the CBP is equal to the upper bound in equation 2 unless otherwise specified; we do this for clarity of exposition since it avoids having to also list the CBP value for each input/output pair of edges. This means that the size of the merged buffer will be assumed to be taken from table 2 for all the illustrative examples we use (since for the MFIR, the CBP is equal to the upper bound in equation 2), unless specified otherwise. Note that none of our results are affected by this assumption; the assumption only applies to examples we use for illustrative purposes.

5.3.2 Merging a chain of buffers

Let $b_i \oplus b_o$ denote the buffer resulting from merging the buffers b_i and b_o, on edges e_i and e_o respectively. Define $|b|$ to be the size of a buffer b. Define the **augmentation function** $A(b_i \oplus b_o)$ to be the amount by which the output buffer b_o has to be augmented due to the merge $b_i \oplus b_o$. That is,

$$A(b_i \oplus b_o) = |b_i \oplus b_o| - |b_o| . \qquad \textbf{(EQ 6)}$$

For OD edge pairs, $|b_o| = I_1p$ and for ID edge pairs, $|b_o| = I_2p$. Hence, table 2, can be rewritten in terms of the augmentation as

Table 3: Augmentation function for MFIR

	OD	ID
$c - p < 0$	0	$I_1 c - I_2 c$
$c - p \geq 0$	$I_2(c - p)$	$I_1 c - I_2 p$

Observation 3: The merge operator is associative.

Proof: Let $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4$ be a chain of four actors, let $e_i = (v_i, v_{i+1}), i = 1, 2, 3$, and let b_i be the respective buffers. Then we have to show that $|(b_1 \oplus b_2) \oplus b_3| = |b_1 \oplus (b_2 \oplus b_3)|$. Consider figure 8,

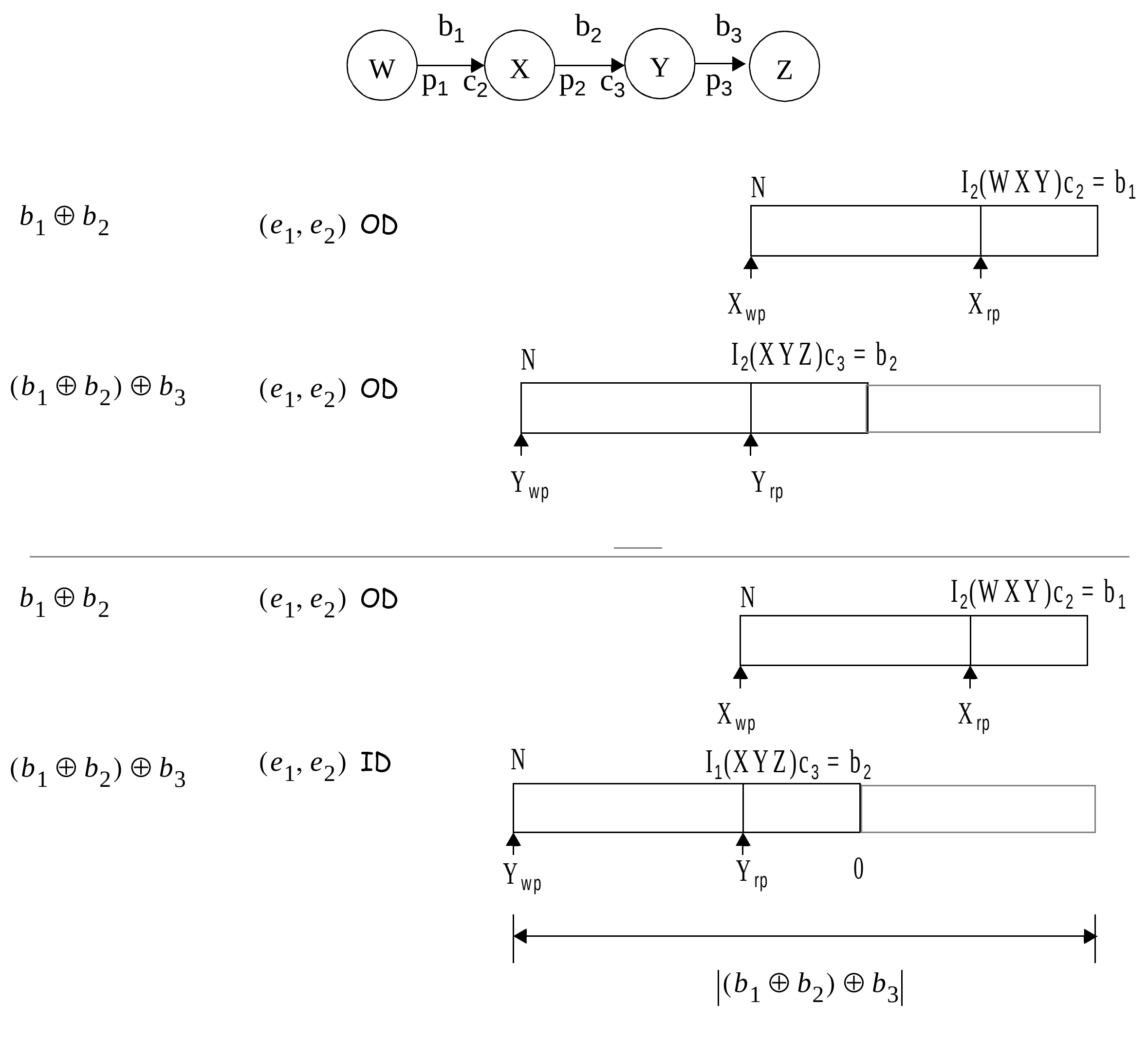


Fig 8. Overlaying three buffers b_1, b_2, b_3 using the association $(b_1 \oplus b_2) \oplus b_3$. Two of the four possibilities are pictured here: (e_1, e_2) is OD and (e_2, e_3) is OD, and (e_1, e_2) is OD and (e_2, e_3) is ID.

where two of the four possibilities for the association order $(b_1 \oplus b_2) \oplus b_3$ are depicted: (e_1, e_2) is OD and (e_2, e_3) is OD, and (e_1, e_2) is OD and (e_2, e_3) is ID. The term $I_2(WXY))$ denotes the I_2 factor defined earlier, for the subtree $T_{W'X} \subseteq T_{WXY'}$, and $I_2(XYZ)$ is the I_2 factor for the subtree $T_{X'Y} \subseteq T_{XYZ'}$. The terms X_{rp}, X_{wp} (Y_{rp}, Y_{wp}) refer to the read pointer of actor X (Y) from buffer b_1 (b_2) and the write pointer of actor X (Y) on buffer b_2 (b_3) respectively. The figure depicts how the three buffers will be merged: first we do the merge $b_1 \oplus b_2$. Then we overlay b_3 with the merged buffer $b_1 \oplus b_2$. Since the overlaying technique puts the write pointer at the leftmost end of the buffer, in both the OD and ID cases, we see that the only difference in overlaying an OD edge pair with another OD edge pair, compared to overlaying an ID edge pair with an OD edge pair, is the location of the read pointer Y_{rp} in the merged buffer $(b_1 \oplus b_2) \oplus b_3$: the read pointer is located at $I_1(XYZ)c_3$ instead of $I_2(XYZ)c_3$. The read pointer Y_{rp} may not need to traverse the entire buffer $b_1 \oplus b_2$, but will traverse some subset of it, all of which is contiguous and to the left of the portion of the buffer read from by X_{rp} as shown. From this overlaying technique, it is clear that a) the technique is correct in that the merged buffer is valid, and does not result in loss of data, and b) when overlaying b_3 with $b_1 \oplus b_2$, it does not matter how big $b_1 \oplus b_2$ is as long as the subset of $b_1 \oplus b_2$ that Y will read from is placed to the left as shown. Hence, it is clear that we could also perform the merge $b_2 \oplus b_3$ first, and then overlay b_1 to the right of $b_2 \oplus b_3$ as shown. Hence, the order in which we perform the merge does not matter since the size of the merged buffer is not dependent on the size of other buffers, and the ordering of the pointers allows contiguous access.

Theorem 2: Let $v_1 \to v_2 \to \ldots \to v_k$, $k > 2$, be a path (a chain of actors and edges) in the SDF graph. Let b_i be the buffer on the output edge of actor v_i, and let S be a given SAS (according to which the b_i are determined). Then,

$$|b_1 \oplus \ldots \oplus b_{k-1}| = \sum_{i=2}^{k-1} A(b_{i-1} \oplus b_i) + |b_{k-1}|. \qquad \textbf{(EQ 7)}$$

Proof: The proof is by induction on k. For $k = 3$, the formula holds because of equation 6. Now suppose it holds for $k-1$, and consider the chain of actors an edges depicted in figure 9. Let

$$|b| = |b_1 \oplus \ldots \oplus b_{k-2}| = \sum_{i=2}^{k-2} A(b_{i-1} \oplus b_i) + |b_{k-2}| = a + |b_{k-2}|, \text{ where}$$

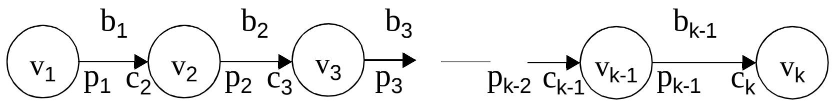


Fig 9. A chain of SDF actors.

$$a = \sum_{i=2}^{k-2} A(b_{i-1} \oplus b_i)$$

Now we want to compute

$$|b'| = |b \oplus b_{k-1}|$$

We show that $|b'| = |b_{k-1}| + A(b_{k-1} \oplus b_{k-2}) + a = |b_{k-1} \oplus b_{k-2}| + |b| - |b_{k-2}|$, proving the theorem. From figures 6 and 7, the overlaid buffer for merging two edges is as depicted in figure 10(a). The write pointer is offset from the read pointer by $|b_i \oplus b_o| - |b_i|$. Now consider merging b_{k-1} with b_{k-2}. Since a buffer of size $|b|$ suffices for the chain with $k-1$ actors, we know that the read pointer for buffer b_{k-2} cannot start to the left of the point denoted b in figure 10(b). The write pointer is offset by a distance of $|b_{k-1} \oplus b_{k-2}| - |b_{k-2}|$; we know that this difference will be enough to implement $b_{k-1} \oplus b_{k-2}$. Hence, the theorem is proved. **QED**.

6 A heuristic for merged cost-optimal SAS

Until now, we have assumed that a SAS was given; we computed the merged costs based on this SAS. In this section, we develop an algorithm to generate the SAS so that the merged cost is minimized. In [11], a DPPO formulation is given for chain-structured SDF graphs that organizes the optimal loop hierarchy for any SAS based on the cost function where every buffer is implemented separately. In this section, we give a DPPO formulation that uses the new, buffer merging cost function developed in the previous section for organizing a good loop hierarchy for a chain-structured graph. However, unlike the result in [11], our formulation for this new cost function is not optimal for reasons we will show below; however, it is still a good heuristic technique to use.

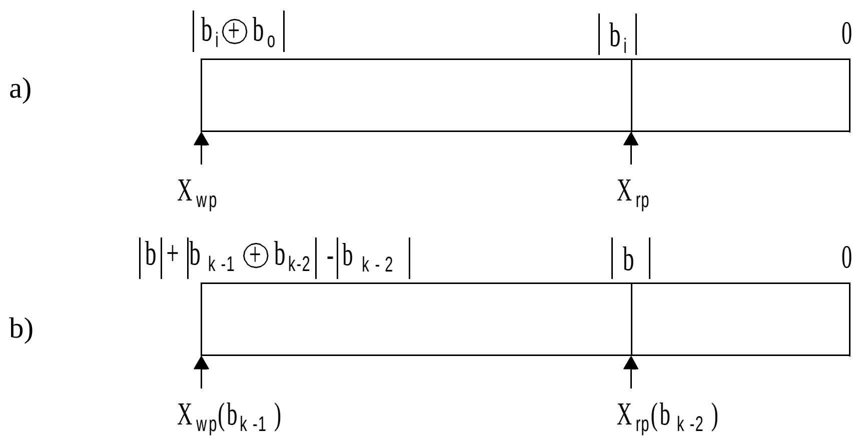


Fig 10. Overlaid buffers in terms of input, output, and merged buffer sizes.

6.1 DPPO formulation

Let $v_i \to v_{i+1} \to \dots \to v_j$ be a sub-chain of actors in the chain-structured SDF graph. The basic idea behind the DPPO formulation is to determine where the split should occur in this chain, so that the SAS S_{ij} for it may be represented as

$$S_{ij} = (i_L S_{ik})(i_R S_{k+1j}) .$$

If S_{ik} and S_{k+1j} are known to be optimal for those subchains, then all we have to do to compute S_{ij} is to determine the $i \le k < j$ where the split should occur; this is done by examining the cost for each of these k. In order for the resulting S_{ij} to be optimal, the problem must have the optimum substructure property: the cost computed at the interfaces (at the split points) should be independent of the schedules S_{ik} and S_{k+1j}. Now, if each buffer is implemented separately, then the cost at the split point is simply the size of the buffer on the edge crossing the split, and this does not depend on what schedule was chosen for the left half (S_{ik}). Hence, the algorithm would be optimal then [11]. However, for the merging cost function, it turns out that the interface cost does depend on what S_{ik} and S_{k+1j} are, and hence this DPPO formulation is not optimal, as shown later. It is a greedy heuristic that attempts to give a good approximation to the minimum. In section 8, we show that on practical SDF systems, this heuristic can give better results than the technique of [11]. In order to compute the interface costs, let the buffers on the edges be $b_i, \dots, b_k, b_{k+1}, \dots, b_{j-1}$. Now suppose that the split occurs at k. That is, actors $v_i, \dots, v_k$ are on the left side of the split. Since we know $\mathrm{cost}(S_{ik})$ and $\mathrm{cost}(S_{k+1j})$ (these are memorized, or stored in the dynamic programming table), we have (by theorem 2) that

$$\mathrm{cost}(S_{ik}) = |b_{k-1}| + A_{ik}, \text{ and } \mathrm{cost}(S_{k+1j}) = |b_{j-1}| + A_{k+1j},$$

where A is the augmentation term. Hence, in order to determine the cost of splitting at k, we have to determine $b_{k-1} \oplus b_k$ and $b_k \oplus b_{k+1}$. Using theorem 2, the total cost is thus given by

$$c_{ij}(k) = \begin{array}{c} \mathrm{cost}(S_{ik}) - |b_{k-1}| + A(b_{k-1} \oplus b_k) \\ + A(b_k \oplus b_{k+1}) + \mathrm{cost}(S_{k+1j}) \end{array} . \qquad \textbf{(EQ 8)}$$

We then choose the k that minimizes the above cost:

$$\mathrm{cost}(S_{ij}) = MIN_{i \le k < j}\{c_{ij}(k)\} .$$

6.2 Computing I_1 and I_2 efficiently

In order to compute $A(b_{k-1} \oplus b_k)$ and $A(b_k \oplus b_{k+1})$, we need the appropriate I_1 and I_2 factors. Observe that $\{e_{k-1}, e_k\}$, the input-output edge pair of actor v_k, is OD, and $\{e_k, e_{k+1}\}$ is ID. Define

$$I_1(v_k, S_{ij})$$

as the I_1 factor of v_k in the schedule S_{ij} assuming that the split in S_{ij} is at k. This is the I_1 factor of v_k when we compute $A(b_{k-1} \oplus b_k)$. Similarly, define

$$I_1(v_{k+1}, S_{k+1j})$$

for v_{k+1} when we compute $A(b_k \oplus b_{k+1})$. Define

$$I_2(v_k, S_{ij}) \text{ and } I_2(v_{k+1}, S_{ij})$$

similarly for the I_2 factors. Define

$$g_{ij} = GCD_{i \le k \le j}\{q(v_k)\},$$

where $q(v_k)$ is the repetitions number for v_k. Finally, define S_{xy}^R to be the right portion of the schedule S_{xy}, and S_{xy}^L to be the left portion. For instance, if the split in S_{xy} happens at $x \le z < y$, then $S_{xy}^L = S_{xz}$.

We will assume that the I_1 and I_2 factors are computed for an R-SAS. By this we mean the following: for any subtree in the schedule tree, the loop factor of the root of the subtree is the GCD of the repetitions of all actors that comprise the leaf nodes of that subtree. We will show later that fully factored schedules have the lowest merged-buffer cost compared to SAS that are not R-schedules. We can compute I_1, I_2 efficiently, by using the following relationships:

Theorem 3:

$$I_1(v_k, S_{ij}) = q(v_k)/g_{ij}, \qquad \textbf{(EQ 9)}$$

$$I_1(v_{k+1}, S_{ij}) = q(v_{k+1})/g_{ij}, \qquad \textbf{(EQ 10)}$$

$$I_2(v_k, S_{ij}) = I_2(v_k, S_{ik}), \qquad \textbf{(EQ 11)}$$

$$I_2(v_{k+1}, S_{ij}) = I_2(v_{k+1}, S_{k+1j}). \qquad \textbf{(EQ 12)}$$

Proof: Consider the schedule tree depicted in figure 11. In the tree on the left, since we assume a fully factored tree, the product of the loop factors of the nodes on the path from the root node (but not including the root node) to the leaf node containing v_k must be $q(v_k)/g_{ik}$. Hence, the I_1 factor of v_k when we merge the trees as shown, will become $q(v_k)/g_{ik} \cdot g_{ik}/g_{ij} = q(v_k)/g_{ij}$. Note that $g_{ij} = GCD(g_{ik}, g_{k+1j})$. This proves equation 9. Equation 10 follows similarly. Equations 11 and 12 follow by observing that $I_2(v_k, S_{ij})$ is the total number of times v_k is invoked in the largest loop not containing v_{k-1}, and this does not depend on how the loop structure for $v_{k+1}, \ldots, v_j$ is organized. **QED**.

Theorem 4:

$$I_2(v_k, S_{ik}) = \begin{cases} I_2(v_k, S_{ik}^R), & \left|S_{ik}^R\right| \geq 2 \\ q(v_k)/g_{ik}, & \left|S_{ik}^R\right| = 1 \end{cases}, \qquad \textbf{(EQ 13)}$$

$$I_2(v_{k+1}, S_{k+1j}) = \begin{cases} I_2(v_{k+1}, S_{k+1j}^L), & \left|S_{k+1j}^L\right| \geq 2 \\ q(v_{k+1})/g_{k+1j}, & \left|S_{k+1j}^L\right| = 1 \end{cases}, \qquad \textbf{(EQ 14)}$$

where $|S|$ for a SAS S is the number of actors in S.

Proof: If the schedule for $v_i, \ldots, v_k$ is such that v_k is in a subschedule with more than one actor, then $I_2(v_k, S_{ik})$ is determined from that subschedule since v_{k-1} will also be part of that subschedule. This proves the top half of equation 13 and 14. If not, then the schedule for $v_i, \ldots, v_k$ contains a split at $k-1$, and v_k is not in any subschedule. Hence, $I_2(v_k, S_{ik}) = q(v_k)/g_{ik}$ in this case, and equation 14 follows similarly. **QED**.

Using these formulas, we can memoize these values as well, by storing them in a matrix each time S_{ij} is determined for some i, j. This way, we don't have to actually build and traverse the partial schedule

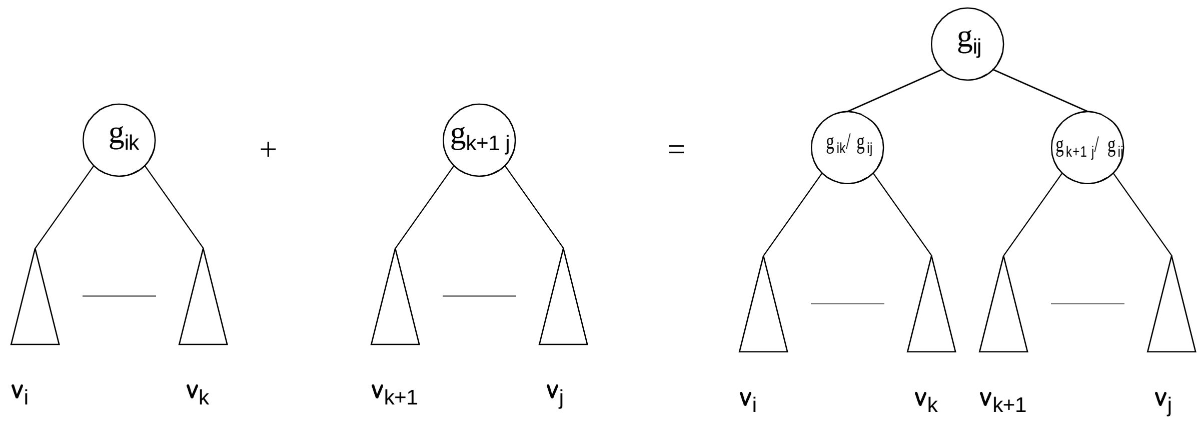


Fig 11. The effect of merging to trees on the I_1 factor of v_k.

tree each time. The entire DPPO algorithm will then have a running time of $O(n^3)$ where n is the number of actors in the chain.

6.3 Factoring

In [12], we show that factoring a SAS by merging loops (in other words, generating nested loops) by the greatest extent possible is not harmful to buffer memory reduction, and that the buffering requirements in a fully factored looped schedule are less than or equal to the requirements in the non-factored loop. Of-course, this result depends on the buffering cost function being used. For example, the result does not, in general, hold under the shared buffer model used in [14]. Happily, this result does hold for the merging cost function, as shown by the following theorem:

Theorem 5: Suppose that $S = (i_L S_{ik})(i_R S_{k+1j})$ is a valid SAS for a chain-structured SDF graph G. Define $\mathrm{cost}(S)$ to be the size of the buffer obtained by merging all the buffers on the edges in S. Then, for any positive integer γ that divides i_L and i_R, the schedule

$$S' = \gamma\left\{\left(\frac{i_L}{\gamma}S_{ik}\right)\left(\frac{i_R}{\gamma}S_{k+1j}\right)\right\}$$

satisfies $\mathrm{cost}(S') \le \mathrm{cost}(S)$.

Proof: First off, note that $\mathrm{cost}(i \cdot S) = \mathrm{cost}(j \cdot S) \quad \forall i, j$ for any schedule S; hence, only the buffer crossing the cut between S_L and S_R is affected. This also means that $\mathrm{cost}(S') = \mathrm{cost}\left(\left(\frac{i_L}{\gamma}S_{ik}\right)\left(\frac{i_R}{\gamma}S_{k+1j}\right)\right)$. Let $S'_{ik} = \frac{i_L}{\gamma}S_{ik}$ and $S'_{k+1j} = \frac{i_R}{\gamma}S_{k+1j}$. Hence, $\mathrm{cost}(S') = \mathrm{cost}((S'_{ik})(S'_{k+1j}))$. By equation 8, we have that

$$\mathrm{cost}(S) = \begin{array}{c} \mathrm{cost}(S_{ik}) - b_{k-1} + A(b_{k-1} \oplus b_k) \\ + A(b_k \oplus b_{k+1}) + \mathrm{cost}(S_{k+1j}) \end{array}.$$

Since $\mathrm{cost}(S_{ik}) = \mathrm{cost}(S'_{ik})$, and b_{k-1} is the same in both schedules S and S', the only variables to consider are $A(b_{k-1} \oplus b_k), A(b_k \oplus b_{k+1})$. The edge pair $\{e_{k-1}, e_k\}$ is OD in both S and S'; hence, we have that

$$A(b_{k-1} \oplus b_k) = \begin{cases} c_k - p_k + |CBP_k| & c_k - p_k < 0 \\ I_2(v_k, \bar{S})(c_k - p_k) + |CBP_k| & c_k - p_k \ge 0 \end{cases},$$

where $\bar{S} \in \{S, S'\}$ for the two cases of interest. Note that the I_1 term cancels because in each case, it is equal to b_k. We also have

$$I_2(v_k, \bar{S}) = \begin{cases} I_2(v_k, S_{ik}) & \bar{S} = S \\ I_2(v_k, S'_{ik}) & \bar{S} = S' \end{cases},$$

and $I_2(v_k, S'_{ik}) = I_2(v_k, S_{ik})$ since the loop factor in front of S_{ik} in S'_{ik} does not affect $I_2(v_k, S'_{ik})$. Therefore, $A(b_{k-1} \oplus b_k)$ is the same in both cases. For $A(b_k \oplus b_{k+1})$, we have

$$A(b_k \oplus b_{k+1}) = \begin{cases} I_1(v_{k+1}, \bar{S})c + I_2(v_{k+1}, \bar{S})(p_{k+1} - c_{k+1}) + c_{k+1} - p_{k+1} + |CBP_{k+1}| - b_{k+1} \\ I_1(v_{k+1}, \bar{S})c_{k+1} + |CBP_{k+1}| - b_{k+1} \qquad c_{k+1} - p_{k+1} \geq 0 \end{cases}.$$

Again, the I_2 term and b_{k+1} terms are identical to both cases ($\bar{S} \in \{S, S'\}$), as are all of the production/consumption/CBP terms. The only term that varies is the I_1 term. We have

$$I_1(v_{k+1}, \bar{S}) = \begin{cases} i_R \cdot I_1(v_{k+1}, S_{k+1j}) & \bar{S} = S \\ \frac{i_R}{\gamma} \cdot I_1(v_{k+1}, S_{k+1j}) & \bar{S} = S' \end{cases}.$$

Since the second case is smaller, we see that the merged cost goes down with factoring, and thus, a fully factored schedule has the lowest merge cost. **QED**.

6.4 Suboptimality of the DPPO formulation

Unfortunately, we cannot prove that the DPPO formulation of section 6.1 is optimal. In order to see this, consider the schedule trees in figure 12. Since the subtrees in the left tree are optimal, we have

$$\text{cost}(S'_L) > \text{cost}(S_L) \text{ and } \text{cost}(S'_R) > \text{cost}(S_R).$$

Assuming that the split between S_L, S_R happens on edge e_k, we have that $\text{cost}(S_L) = b_{k-1} + A_L$, where A_L is the cumulative augmentation term for the merged buffers in S_L. Similarly, we have $\text{cost}(S'_L) = b'_{k-1} + A'_L$. We also have $b_{k-1} = I_2(v_k, S_L)c_k$ and $b'_{k-1} = I_2(v_k, S'_L)c_k$. If $c_k - p_k > 0$, merging b_{k-1} and b_k gives us a cost of $b_k + A_L + A(b_{k-1} \oplus b_k)$. The two cases then become

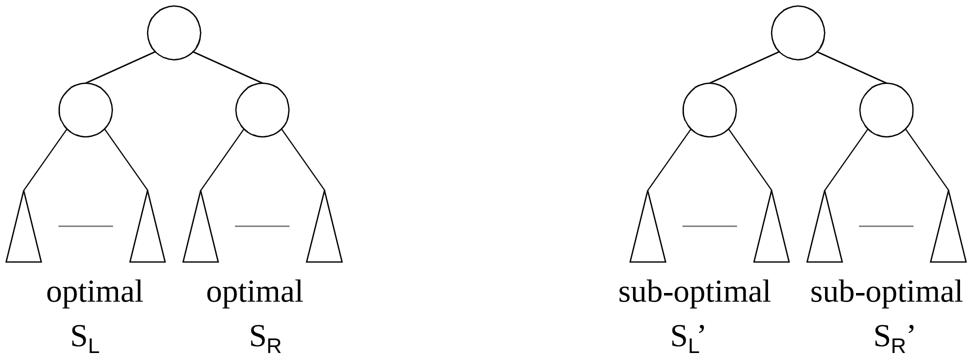


Fig 12. Optimal and suboptimal schedule trees.

$$b_k + A'_L + I_2(v_k, S'_L)(c_k - p_k) \text{ and } b_k + A_L + I_2(v_k, S_L)(c_k - p_k).$$

If $A'_L + I_2(v_k, S'_L)(c_k - p_k) > A_L + I_2(v_k, S_L)(c_k - p_k)$ is satisfied, then optimal subschedules will lead to optimal schedules, but there is no guarantee that the above relation is satisfied; optimality of the subschedules only guarantees that $A'_L + I_2(v_k, S'_L)c_k > A_L + I_2(v_k, S_L)c_k$.

7 Acyclic graphs

In this section, we extend the merging techniques to arbitrary, delayless, acyclic SDF graphs. The techniques we develop here can easily be extended to handle graphs that have delays, as discussed in Section 5. SASs for SDF graphs that contain cycles can be constructed in an efficient and general manner by using the loose interdependence scheduling framework (LISF) [2]. The LISF operates by decomposing the input graph G into a hierarchy of acyclic SDF graphs. Once this hierarchy is constructed, any algorithm for scheduling acyclic SDF graphs can be applied to each acyclic graph in the hierarchy, and the LISF combines the resulting schedules to construct a valid SAS for G. Thus, the LISF provides an efficient mechanism by which the techniques developed here can be applied to general (not necessarily acyclic) topologies.

When acyclic graphs are considered, there are two other dimensions that come into play for designing merging algorithms. The first dimension is the choice of the topological ordering of the actors; each topological ordering leads to a set of SASs. This dimension has been extensively dealt with before in [2], where we devised two heuristic approaches for determining good topological orderings. While these heuristics were optimized for minimizing the buffer memory cost function where each buffer is implemented separately, they can be used with the new merged cost function as well. We leave for future work to design better heuristics for the merged cost function, if it is possible.

The second dimension is unique to the merge cost function, and is the issue of the set of paths that buffers should be merged on. In other words, given a topological sort of the graph, and a nested SAS for this graph, there still remains the issue of what paths buffers should be merged on. For the chain-structured graphs of the previous section, there is only one path, and hence this is not an issue. Since an acyclic graph can have an exponential number of paths, it does become an issue when acyclic graphs are considered. In the following sections, we develop two approaches for determining these paths. The second approach is a bottom up approach that combines lifetime analysis techniques from [14] and the merging approach to generate implementations that arguably extract the maximum benefit of both approaches. However, a drawback of this approach is that it is of high complexity and is slow. The first approach does not use lifetime analysis techniques, and instead determines the optimum set of paths along which buffers should be merged. The algorithm we give is optimal in the sense that for a given topological ordering and SAS, our algorithm will determine the lowest merge cost implementation when buffers are merged in a linear order along the paths. Yet another dimension can be introduced by not merging the buffers linearly; this is captured by clustering as we show later.

7.1 Path covering

Determining the best set of paths to merge buffers on can be formulated as a path covering problem. Essentially, we want a disjoint set of paths Ψ such that each edge in the graph is in exactly one path in Ψ. The total buffering cost is then determined by merging the buffers on the edges in each path, and summing the resulting costs.

Example 2: Consider the graph shown in figure 13. The schedule tree is shown on the right, and represents the SAS $5A\ 2(3(2B\ 3C)\ 2D)$. There are two possible ways of merging buffers in this graph: $b_1 \oplus b_2 \oplus b_3 + b_4$ and $b_1 \oplus b_2 + b_4 \oplus b_3$. These correspond to the paths $\{(AB, BC, CD), (AC)\}$ and $\{(AB, BC), (AC, CD)\}$. The non-merged costs for the buffers in each edge, for the schedule shown, are given by $b_1 = 60$, $b_2 = 6$, $b_3 = 36$, and $b_4 = 90$. Thus, if each of these were to be implemented separately, the total buffering cost would be $60 + 6 + 36 + 90 = 192$. It can be verified that $b_1 \oplus b_2 \oplus b_3 + b_4 = 180$, and $b_1 \oplus b_2 + b_4 \oplus b_3 = 168$. Hence, the better set of paths to use for this example is $\{(AB, BC), (AC, CD)\}$.

Given a directed graph (digraph) G, an **edge-oriented** path is a sequence of edges $e_1, e_2, \ldots, e_n$ such that $src(e_i) = snk(e_{i-1})$ for each $i = 2, 3, \ldots, n$. A **node-oriented** path is a sequence of nodes $v_1, \ldots, v_n$ such that (v_i, v_{i+1}) is an edge in the graph for each $i = 1, \ldots, n-1$. The **edge-set of a node-oriented path** $v_1, \ldots, v_n$ is the set of edges (v_i, v_{i+1}). An **edge-oriented path cover** χ is defined as a set of edge-disjoint edge-oriented paths whose union is the entire edge set of G. A **node-oriented path cover** Ψ is defined as a set of node-disjoint node-oriented paths whose union is the entire node set of G. In other words, each node in G appears in exactly one node-oriented path $p \in \Psi$. The **edge-set of a node-oriented path cover** is the union of the edge-sets of each node-oriented path in the cover.

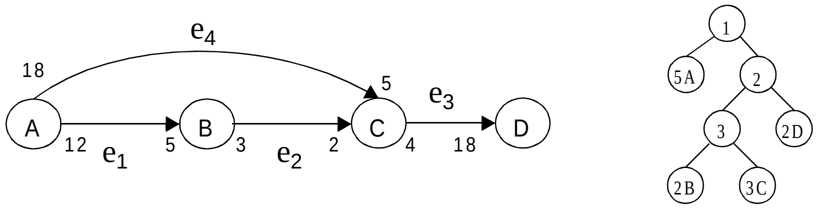


Fig 13. An example to show the variation of the merge cost with path selection.

For an SDF graph G, define the **buffer cost of an edge-oriented path** $e_1, e_2, \ldots, e_n$ as $|b_1 \oplus \ldots \oplus b_n|$, where b_i is the buffer on edge e_i. Define the **buffer cost of an edge-oriented path cover** χ as the sum of the buffer costs of the paths in the cover.

Definition 5: The PATH SELECTION PROBLEM FOR BUFFER MERGING (**PSPBM**) in an acyclic SDF graph is to find an edge-oriented path cover of minimum buffer cost.

In order to solve the path selection problem, we first derive a weighted, directed MERGE GRAPH from the SDF graph $G = (V, E)$. The **MERGE GRAPH** MG is defined as $MG = (V_{MG}, E_{MG}, w)$, where

$$V_{MG} = \{v_e | e \in E\} \cup \{s_e \mid e \in E\},$$

$$E_{MG} = \{(v_{e_1}, v_{e_2}) \mid e_1 \in E, e_2 \in E, snk(e_1) = src(e_2)\} \cup \{(v_e, s_e) \mid e \in E\},$$

$$w((v_{e_1}, v_{e_2})) = A(b(e_1) \oplus b(e_2)), \text{ and}$$

$$w((v_e, s_e)) = |b(e)|.$$

The buffer on an edge e in the SDF graph G is denoted by $b(e)$. Figure 14 shows the MERGE GRAPH for the SDF graph in figure 13. The nodes of type s_e are called **S-type nodes**.

Fact 2: If the SDF graph is acyclic, the associated MERGE GRAPH is also acyclic.

Given a weighted digraph G, the **weight of a path** p in G, denoted as $w(p)$ is the sum of the weights on the edges in the path. The **weight of a path cover** Ψ, denoted as $w(\Psi)$ is the sum of the path weights of the paths in Ψ.

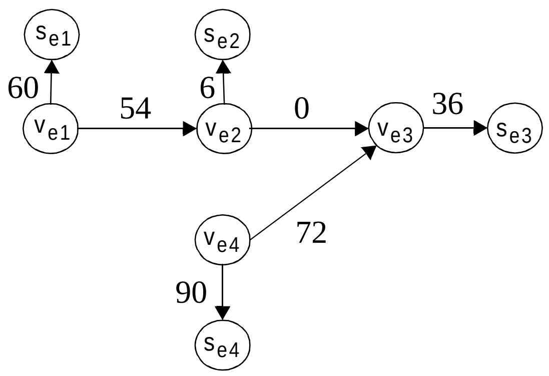


Fig 14. The MERGE GRAPH for the SDF graph in figure 13.

We define a **maximal path cover** for the MERGE GRAPH as a node-oriented path cover Ψ such that each path $p \in \Psi$ ends in an S-type node. A **minimum weight maximal path cover (MWMPC)** Ψ^* is a maximal path cover of minimum weight.

Definition 6: The **MWMPC problem** for MERGE GRAPHs is to find an MWMPC.

Given an MWMPC Ψ for the MERGE GRAPH MG, for each path p in the MWMPC, replace each (non S-type) node v_e in p by the corresponding edge e in the SDF graph G to get an edge-oriented path q in G. Let χ be the set of paths q. Note that we do not have any edges corresponding to S-type nodes in MG. Then we have the following obvious result:

Lemma 2: The set of edge-oriented paths χ constructed above is a solution to the PSPBM problem; that is, χ is an edge-oriented path cover of minimum buffer cost for the SDF graph G.

Proof: Since Ψ is an MWMPC, let $p = \{v_{e_1}, \ldots, v_{e_n}, s_{e_n}\}$ be a node-oriented path in Ψ. The corresponding set of edges $\{e_1, \ldots, e_n\}$ is an edge-oriented path $q = \{e_1, \ldots, e_n\}$ in SDF graph G: by definition of the MERGE GRAPH, (v_{e_i}, v_{e_j}) is an edge iff $snk(e_i) = src(e_j)$ in G. Since each node v_e appears once in Ψ, each edge e appears once in χ, and thus χ is an edge-oriented path cover for G. The weight $w(p)$, $p \in \Psi$ is identical to the buffer cost $bc(q)$, $q \in \chi$:

$$w(p) = \sum_{i=1}^{n-1} w((v_{e_i}, v_{e_{i+1}})) + w((v_{e_n}, s_{e_n})) = \sum_{i=1}^{n-1} A(b(e_i) \oplus b(e_{i+1})) + \left|b_{e_n}\right| = \left|b_{e_i} \oplus \ldots \oplus b_{e_n}\right|.$$

Since Ψ is an MWMPC, it follows that χ is an edge-oriented path cover of minimum buffer cost. **QED**.

For example, in figure 14, the MWMPC is given by $\{s_{e_1}, (v_{e_1} \to v_{e_2} \to s_{e_2}), (v_{e_4} \to v_{e_3} \to s_{e_3}), s_{e_4}\}$, and it can be verified easily that this corresponds to the optimal buffer merge paths shown in example 2.

In [10], Moran et al. give a technique for finding maximum weight path covers in digraphs. We modify this technique slightly to give an optimum, polynomial time algorithm for finding an MWMPC in a MERGE GRAPH. Given a weighted, directed acyclic graph $G = (V, E, w)$, with $V = \{v_1, \ldots, v_n\}$, we first derive the following weighted bipartite graph $G_B = (X, Y, E_B, w_B)$:

$$X = \{x_i \mid v_i \in V\}$$

$$Y = \{y_i \mid v_i \in V\}$$

$$E_B = \{(x_i, y_j) \mid (v_i, v_j) \in E\}$$

$$w_B((x_i, y_j)) = W^* - w((v_i, v_j)) \qquad \forall (v_i, v_j) \in E \text{, where } W^* = MAX_{(v_i, v_j) \in E}(w((v_i, v_j))) + 1$$

Figure 15 shows a weighted digraph and the corresponding bipartite graph.

A **matching** M is a set of edges such that no two edges in M share an endpoint. The weight of a matching is the sum of the weights of the edges in the matching. A **maximum weight matching** is a matching of maximum weight. Maximum weight matchings in bipartite graphs can be found in polynomial time ($O(|V_B|^2 \cdot \log(|V_B|))$); for example, using the “Hungarian” algorithm [15].

Given a matching M in G_B, define $E_\Psi = \{(v_i, v_j) \mid (x_i, y_j) \in M\}$. That is,

$$(v_i, v_j) \in E_\Psi \Leftrightarrow (x_i, y_j) \in M \qquad \textbf{(EQ 15)}$$

Lemma 3: The graph (V, E_Ψ) consists of only directed paths, meaning that if $(v_i, v_j) \in E_\Psi$, then there is no $v_k \in V, k \neq i, k \neq j$ such that $(v_k, v_j) \in E_\Psi$ or $(v_i, v_k) \in E_\Psi$.

Proof: Suppose that $(v_i, v_j) \in E_\Psi$. This means that $(x_i, y_j) \in M$. Then we cannot have $v_k \neq v_i$ such that $(v_k, v_j) \in E_\Psi$. Indeed, if $(v_k, v_j) \in E_\Psi$ were true, then this would mean that $(x_k, y_j) \in M$ and would contradict M being a matching. Similarly, we cannot have $v_k \neq v_j$ such that $(v_i, v_k) \in E_\Psi$. Finally, we cannot have a directed cycle because $(V, E_\Psi) \subseteq (V, E)$, and (V, E) is acyclic.

Lemma 4: Let the graph $G = (V, E)$ be a MERGE GRAPH, and let M be a maximum weight matching in G_B. Every path in E_Ψ includes an S-type node.

Proof: Suppose it did not, and there were some path in E_Ψ that did not include an S-type node. Let this path be $v_1 \rightarrow \ldots \rightarrow v_k$, where v_k is not an S-type node. We have that $(x_i, y_{i+1}) \in M \quad \forall i = 1, \ldots, k-1$. Now consider the edge $(x_k, y_{s,k})$, where $y_{s,k}$ is the node corre-

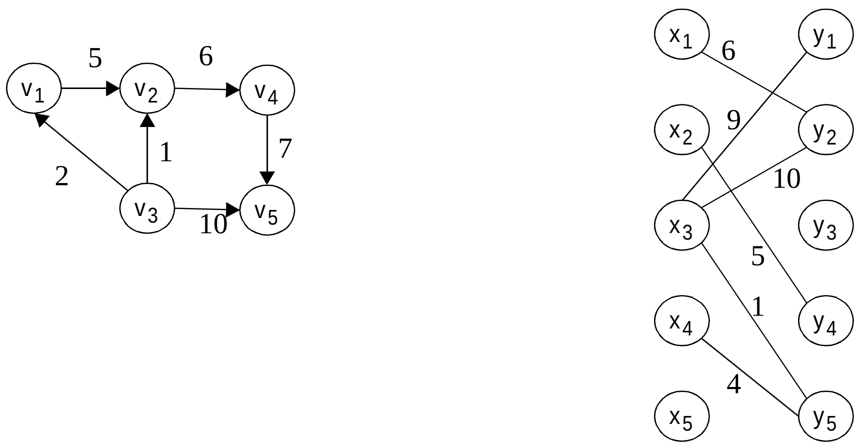


Fig 15. The bipartite graph derived from a weighted digraph.

sponding to an S-type node from G. Since this edge is not in M, we can add it to M and get a larger weight matching. Note that adding this edge to M is possible because S-type nodes have only one incoming edge; hence, there is no other x_j such that $(x_j, y_{s,k})$ is an edge in G_B. Also, there is no other edge of the type (x_k, y_p) in the matching since v_k was the last node in a connected path by supposition.

Lemma 5: Every node-oriented maximal cover in the MERGE GRAPH has the same number of edges in its edge set.

Proof: This follows because of the requirement that an S-type node be contained in every path. This ensures that each non-S-type node has exactly one of its output edges in the cover. S-type nodes do not have any output edges, so they do not contribute any edges to the cover.

Theorem 6: If M^* is a maximum weight matching in G_B, and E_Ψ is as defined earlier, then $\Psi^* = Co(V, E_\Psi)$ is an MWMPC for the MERGE GRAPH G, where $Co(V, E_\Psi)$ denotes the connected components of (V, E_Ψ).

Proof: Suppose that there is another maximal cover Ψ' of lesser weight than Ψ^*. Corresponding to Ψ' there is a matching M' defined as in equation 15. We have

$$w(\Psi') = \sum_{(v_i, v_j) \in \Psi'} w((v_i, v_j)) = \sum_{(x_i, y_j) \in M'} w((v_i, v_j))$$

and

$$w(M') = |M'|W^* - \sum_{(x_i, y_j) \in M'} w((v_i, v_j))$$

where $|M|$ is the number of edges in matching M. Similarly, we have for the maximum weight matching and MWMPC:

$$w(\Psi^*) = \sum_{(x_i, y_j) \in M^*} w((v_i, v_j)) = \sum_{(v_i, v_j) \in \Psi^*} w((v_i, v_j))$$

and

$$w(M^*) = |M^*|W^* - \sum_{(x_i, y_j) \in M^*} w((v_i, v_j))$$

Since Ψ' is supposed to be of lesser weight than Ψ^*, we have

$$\sum_{(x_i, y_j) \in M'} w((v_i, v_j)) < \sum_{(x_i, y_j) \in M^*} w((v_i, v_j)) \text{, or}$$

$$- \sum_{(x_i, y_j) \in M'} w((v_i, v_j)) > - \sum_{(x_i, y_j) \in M^*} w((v_i, v_j)) . \qquad \textbf{(EQ 16)}$$

```
Procedure determineMergePaths(SDF Graph G)
        $G_E \leftarrow$ MergeGraph(G)
        $G_B \leftarrow$ BipartiteGraph($G_E$)
        $M^* \leftarrow$ maxMatching($G_B$)
        $(v_i, v_j) \in \Psi^* \Leftrightarrow (x_i, y_j) \in M^*$
        return $\Psi^*$
```

Fig 16. Procedure for computing the optimum set of paths along which buffers should be merged.

Since Ψ' and Ψ^* are both maximal covers, they have the same number of edges by lemma 5. Since the induced matching has the same number of edges as the cover, we have $|M^*| = |M'|$. Thus, equation 16 implies that $w(M') > w(M^*)$, contradicting the fact that M^* is the maximum weight matching. **QED**.

The algorithm for finding an optimal set of paths along which to merge buffers is summarized in figure 16.

7.1.1 Running time

As already mentioned, the matching step takes $O(|V_b|^2 \cdot \log(|V_b|))$, where V_b is the set of nodes in the bipartite graph. Since $|V_b| = |X| + |Y| = 2 \cdot |V_{MG}|$, where X, Y are the node sets in the bipartite graph, and V_{MG} is the node set in the MERGE GRAPH, we have $|V_{MG}| = 2 \cdot |E|$ and $|V_b| = 4 \cdot |E| = O(|E|)$, where E is the set of edges in the SDF graph. The construction of the MERGE GRAPH takes time $O(|V| \cdot \log(|V|) + |E|)$ if the SDF graph has actors with constant in-degree and out-degree. The $\log(|V|)$ comes from computing the buffer merges to determine the weights in the MERGE GRAPH; recall that since the SAS is given now, computing a merge requires traversal of the schedule tree, and can be done in time $\log(|V|)$ on average if the tree is balanced. If the tree is not balanced, then the merge computation could take $O(|V|)$ time, meaning that the MERGE GRAPH construction takes $O(|V|^2 + |E|)$ time. The bipartite graph also takes time $O(|E|)$. Hence, the overall running time is dominated by the matching step, and takes time $O(|E|^2 \cdot \log(|E|))$, or

$$O(|V|^2 \cdot \log(|V|))$$

if the SDF graph is sparse.

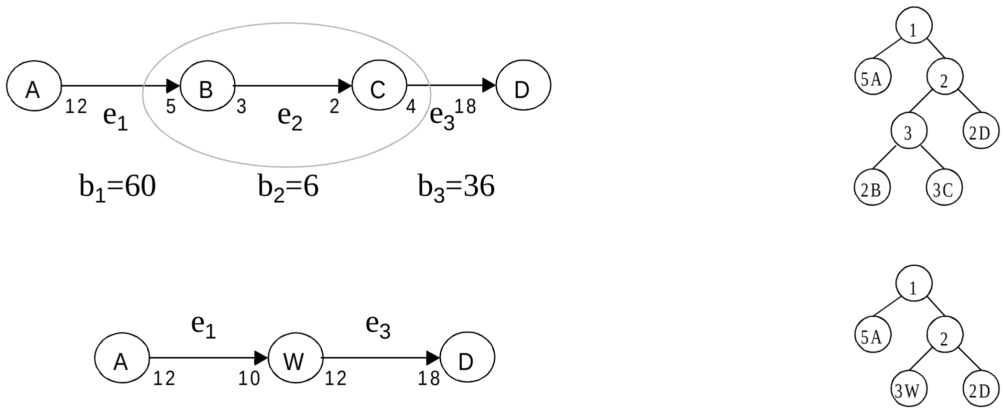


Fig 17. The effect of clustering on buffer merging.

7.1.2 Clustering

While the buffer merging technique as developed results in significant reductions in memory requirements, even more reduction can be obtained if other optimizations are considered. The first of these is the use of clustering. Until now, we have implicitly assumed that the buffers that are merged along a chain are overlaid in sequence. However, this may be a suboptimal strategy since it may result in a fragmented buffer where lot of storage is wasted. Hence, the optimization is to determine the sub-chains along a chain where buffers should be profitably merged, and not to blindly merge all buffers in a chain. This can be captured via clustering, where the cluster will determine the buffers that are merged. For instance, consider the SDF graph in figure 17. If we merge the buffers in the top graph, we get a merged buffer of size 90. However, if we merge the two edges in the clustered graph at the bottom, where actors B and C have been clustered together into actor W, we get a merged buffer of size 66. The edge between B and C is implemented separately, and it requires 6 storage units. Hence the total buffering cost goes down to 72. The reason that this happens is shown in figure 18. The buffer of size 6 between the two larger buffers fragments the overall buffer and results in some space being wasted. The clustering removes this small buffer and merges only the two larger ones, enabling more efficient use of storage.

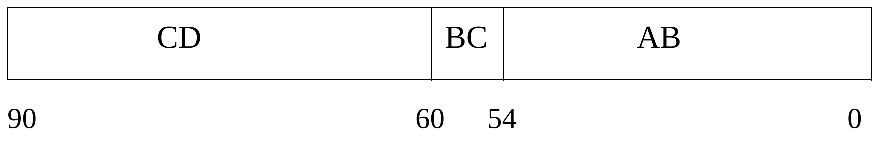


Fig 18. Fragmentation due to merging all buffers in series.

The above optimization can be incorporated into the path covering algorithm by introducing transitive edges in the MERGE GRAPH construction. Instead of just having edges between nodes that correspond to input/output edges of the same actor in the SDF graph, we introduce edges between two nodes if they correspond to edges on some directed path in the SDF graph. Figure 19 shows the MERGE GRAPH for the SDF graph in figure 17.

7.2 A bottom-up approach

Now we describe another technique for determining merge paths that also combines lifetime analysis techniques from [14]. Briefly, the lifetime analysis techniques developed in [14] construct an SAS optimized using a particular shared-buffer model that exploits temporal disjointedness of the buffer lifetimes. The method then constructs an intersection graph that models buffer lifetimes by nodes and edges between nodes if the lifetimes intersect in time. FirstFit allocation heuristics [13] are then used to perform memory allocation on the intersection graph. The shared buffer model used in [14] is useful for modeling the sharing opportunities that are present in the SDF graph as a whole, but is unable to model the sharing opportunities that are present at the input/output buffers of a single actor. The model has to make the conservative assumption that all input buffers are simultaneously live with all output buffers of an actor while the actor has not fired the requisite number of times in the periodic schedule. This means that input/output buffers of a single actor cannot be shared under this model. However, the buffer merging technique developed in this paper models the input/output edge case very well, and is able to exploit the maximum amount of sharing opportunities. However, the merging process is not well suited for exploiting the overall sharing opportunities present in the graph, as that is better modeled by lifetime analysis. Hence, the bottom-up approach we give here combines both these techniques, and allows maximum exploitation of sharing opportunities at both the global level of the overall graph, and the local level of an individual input/output buffer pair of an actor.

The algorithm is stated in figure 20. It basically makes several passes through the graph, each time merging a suitable pair of input/output buffers. For each merge, a global memory allocation is performed

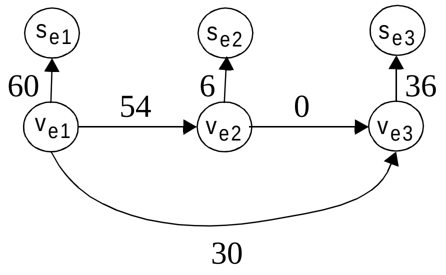


Fig 19. Adding transitive edges to the MERGE GRAPH to enable more optimized merging.

using the combined lifetime of the merged buffer. That is, the start time of the merged buffer is the start time of the input buffer, and the end time is the end time of the output buffer (the procedure `changeIntersectionGraph` performs this). If the allocation improves, then the merge is recorded (procedure `recordMerge`). After examining each node and each pair of input/output edge pairs, we determine whether the best recorded merge improved the allocation. If it did, then the merge is performed (procedure `mergeRecorded`), and another pass is made through the graph where every node and its input/output edge pairs is examined. The algorithm stops when there is no further improvement.

```
Procedure mergeBottomUp(SDF Graph $G$, SAS $S$)

$I \leftarrow$ computeIntersectionGraph($G, S$)
$cur_{best} \leftarrow allocate(I)$
$iter_{best} \leftarrow cur_{best}$

while (true)
    for each node $v \in G$                                   (1)
        for each input edge $e_i$ of $v$                     (2)
            for each output edge $e_o$ of $v$                 (3)
                $b \leftarrow b_i \oplus b_o$                 (4)
                $I \leftarrow$ changeIntersectionGraph($S, b$)
                $m \leftarrow allocate(I)$
                if ($m < iter_{best}$)
                    recordMerge($b, I, m$)
                    $iter_{best} \leftarrow m$
                fi
                restore($I, b_i, b_o, S$)
            end for
        end for
    end for
    if ($iter_{best} < cur_{best}$)
        $cur_{best} \leftarrow iter_{best}$
        mergeRecorded()
    else
        break
    fi
end while
```

Fig 20. A bottom-up approach that combines buffer merging with lifetime analysis.

7.2.1 Running time analysis

The loops labelled (1), (2), and (3) take

$$\sum_{i=1}^{|V|} indeg(v_i) \cdot outdeg(v_i) \qquad \textbf{(EQ 17)}$$

steps, where $indeg(v)$ is the in-degree of actor v and $outdeg(v)$ is the out-degree of actor v. In the worst possible case, we can show that this sum is $O(|V|^3)$, assuming a dense, acyclic graph. If the graph is not dense, and the in- and out- degrees of the actors are bounded by pre-defined constants, as they usually are in most SDF specifications, then equation 17 would be $O(|V|)$. The merging step in line (4) can be pre-computed and stored in a matrix since merging a buffer with a chain of merged buffers just involves merging the buffer at the end of the chain and summing the augmentation. This precomputation would store the results in an $|E| \times |E|$ matrix, and would take time $O(|E|^2 \cdot \log|V|)$ in the average case, and $O(|E|^2 \cdot |V|)$ time in the worst case. So line (4) would end up taking a constant amount of time since the precomputation would occur before the loops. The intersectionGraph procedure can take $O(|E|^2)$ time in the worst case. While this could be improved by recognizing the incremental change that actually occurs to the lifetimes, it is still hampered by the fact that the actual allocation heuristic still takes time $O(|E|^2)$. The overall while loop can take $O(|E|)$ steps since each edge could end up being merged. Hence, the overall running time, for practical systems, is $O(|V| \cdot |E|^3)$ which is

$$O(|V|^4)$$

for sparse graphs. Improvement, if any, can be achieved by exploring ways of implementing the FirstFit heuristic to work incrementally (so that it does not take $O(|E|^2)$); however, this is unlikely to be possible as the merged buffer will have a different lifetime and size, and the allocation has to be redone from scratch each time.

8 Experimental results

8.1 CD-DAT

Consider the SDF representation of the CD-DAT sample rate conversion example from [11], shown in figure 21. The best schedule obtained for this graph in [11], using the non-merged buffering model, has a cost of 260. If we take this SAS, and merge the buffers, then the cost goes down to 226. Applying the new DPPO formulation based on the merging cost, gives a different SAS, having a merged cost of 205. This represents a reduction of more than 20% from previous techniques.

8.2 Homogenous SDF graphs

Unlike the techniques in [11][2], the buffer merging technique is useful even if there are no rate changes in the graph. For instance, consider a simple, generic image-processing system implemented using SDF shown in figure 22. This graph has a number of pixelwise operators that can be considered to have a CBP of 0 for any input-output edge pair. The graph is homogenous because one token is exchanged on all edges; however, the token can be a large image. Most previous techniques, and indeed many current block-diagram code-generators (SPW, Ptolemy, DSPCanvas) will generate a separate buffer for each edge, requiring storage for 8 image tokens; this is clearly highly wasteful since it can be almost seen by inspection that 3 image buffers would suffice. Our buffer merging technique gives an allocation of 3 buffers as expected. In particular, for the example below, we can choose the path to be from A to $Disp$ and apply the merge along that path. Applying the bottom-up approach will reduce this further to 2 buffers since the lifetime analysis will show that C's output can reuse the location that B used.

8.3 A number of practical systems

Table 4 shows the results of applying the bottom-up technique to a number of practical SDF systems. These are all multirate filterbank systems, with the exception of the last one which is a satellite receiver implementation from [17]. The filterbank examples are denoted using the following notation: "qmf23_5d" means that the system is a complete construction-reconstruction system of depth 5; that is, 32 channels. The "23" denotes that a 1/3-2/3 split is used for the spectrum; that is, the signal is recursively divided by taking 1/3 of the spectrum (low-pass) and passing 2/3 of it as the high-pass component. The "qmf12_xd" denote filter banks where a 1/2-1/2 split is used, and the "qmf235_xd" systems denote filterbanks where a 2/5-3/5 split is used.Figure 23(b) shows the "qmf12_3d" system. The rate-changing actors in this system are polyphase FIR filters for which the CBP parameter is obtained using equation 3. The columns named botUp(R) and botUp(A) give the results of applying the bottom-up algorithm on topological orderings generated by the RPMC and APGAN heuristics respectively. The column name "bestShrd" gives

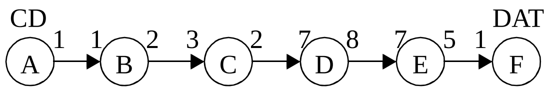


Fig 21. The CD-DAT sample rate converter.

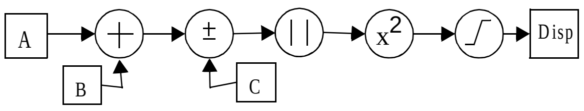


Fig 22. An image processing flow with pixelwise operators.

the best result of applying the lifetime-analysis algorithms from [14]. The “best NonShrd” column contains the best results obtained under the non-shared buffer models, using the algorithms in [2]. The last column gives the percentage improvement of the better of the “botUp(R)” and “botUp(A)” columns compared to the “bestShrd” column; that is, the improvement of the combined buffer merging and lifetime analysis approach compared to the pure lifetime approach. As can be seen, the improvements in memory requirements averages 39% over these examples, and is as high as 54% in one case.

We have not yet tested the top-down technique of section 7.1. We will perform these experiments and report it in the future.

Table 4. Bottom up technique applied to a number of practical systems

Systems	botUp(R)	botUp(A)	bestShrd	bestNonshrd	% Impr.
nqmf23_4d	74	126	132	209	44
qmf23_2d	13	21	22	60	41
qmf23_3d	32	63	63	173	49
qmf23_5d	245	459	492	1271	50
qmf12_2d	9	11	9	34	0
qmf12_3d	11	25	16	78	31
qmf12_5d	30	103	58	342	48
qmf235_2d	30	45	55	122	45
qmf235_3d	110	235	240	492	54

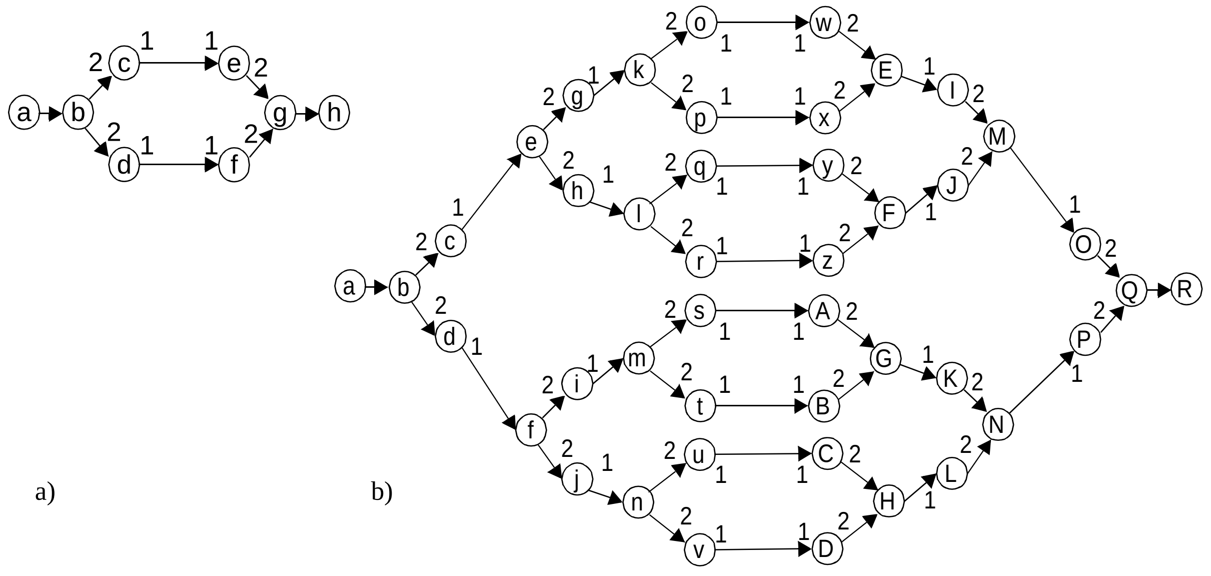


Fig 23. SDF graph for a two-sided filterbank. a) Depth 1 filterbank, b) Depth 3 filterbank. The produced/consumed numbers not specified are all unity.

Table 4. Bottom up technique applied to a number of practical systems

Systems	botUp(R)	botUp(A)	bestShrd	bestNonshrd	% Impr.
qmf235_5d	3790	4500	5690	8967	33
satrec	961	720	991	1542	27

9 Conclusion

Earlier work on SDF buffer optimization has focused on the separate buffer model, and the lifetime model, in which buffers cannot share memory space if any part of the buffers simultaneously contain live data. Our work on buffer merging in this paper has formally introduced a third model of buffer implementation in which input and output buffers can be overlaid in memory if subsets of the buffers have disjoint lifetimes. The technique of buffer merging is able to encapsulate the lifetimes of tokens on edges algebraically, and use that information to develop near-optimal overlaying strategies. While the mathematical sophistication of this technique is especially useful for multirate DSP applications that involve numerous input/output buffer accesses per actor invocation, a side benefit is that it is highly useful for homogenous SDF graphs as well, particularly those involving image and video processing systems since the savings can be dramatic. We have given an analytic framework for performing buffer merging operations, and developed a dynamic programming algorithm that is able to generate loop hierarchies that minimize this merge cost function for chains of actors.

For general acyclic graphs, we have developed two algorithms for determining the optimal set of buffers to merge. The first of these techniques is an innovative formulation using path-covering for determining a provably optimal set of paths (under certain assumptions) on which buffers should be merged. Since this technique is a pure buffer-merging technique, and does not use lifetime analysis, it is faster and might be useful in cases where fast compile times are especially important. However, we leave for future work to provide a comprehensive experimental study of this theoretically interesting algorithm. The second of these techniques, the bottom-up merging algorithm, combines merging and lifetime analysis. Our experiments show large improvements over the separate-buffer and lifetime-based implementations; in particular, reductions of 39% on average on a number of practical systems.

As mentioned before, lifetime-based approaches break down when input/output edges are considered because they make the conservative assumption that an output buffer becomes live as soon as an actor begins firing, and that an input buffer does not die until the actor has finished execution. This conservative assumption is made in [14] primarily to avoid having to pack arrays with non-rectangular lifetime profiles; if the assumption is relaxed, we would get a jagged, non-rectangular lifetime profile, and this could in theory be packed to yield the same memory consumption requirements as the buffer merging technique. However, packing these non-rectangular patterns efficiently is significantly more difficult (as shown by the exponential worst-case complexity of the techniques in [5]), and moreover, it still does not take into account the very fine-grained production and consumption pattern modeled by the CBP parameter. Hence, the buffer merging technique finesses the problem of packing arrays that have non-rectangular lifetime profiles by providing an exact, algebraic framework that exploits the particular structure of SDF graphs and single appearance looped schedules. This framework can then be used with the lifetime-based approach of [14] efficiently to get significant reductions in buffer memory usage.

Buffer merging does not render separate-buffers or lifetime-based buffer sharing obsolete. Separate buffers are useful for implementing edges that contain delays efficiently. Furthermore, they provide a tractable cost function with which once can rigorously prove useful results on upper bound memory requirements [2]. Lifetime-based sharing is a dual of the merging approach, as mentioned already, and can be fruitfully combined with the merging technique to develop a powerful hybrid approach that is better than either technique used alone, as we have demonstrated with the algorithm of Section 7.2.

10 References

[1] M. Ade, R. Lauwereins, J. Peperstraete, “Implementing DSP Applications on Heterogeneous Targets Using Minimal Size Data Buffers,” IEEE Wkshp. on Rapid Sys. Prototyping, 1996.

[2] S. S. Bhattacharyya, P. K. Murthy, E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer, 1996.

[3] S. S. Bhattacharyya, P. K. Murthy, “The CBP Parameter—a Useful Annotation for SDF Compilers,” Tech report UMIACS-TR-99-56, University of Maryland Institute for Advanced Computer Studies, College Park, MD 20742, http://www.cs.umd.edu/TRs/TRumiacs.html, September 1999.

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

[5] E. De Greef, F. Catthoor, H. De Man, “Array Placement for Storage Size Reduction in Embedded Multimedia Systems,” Intl. Conf. on Application Specific Systems, Architectures, and Processors, 1997.

[6] R. Lauwereins, P. Wauters, M. Ade, and J. A. Peperstraete, “Geometric Parallelism and Cyclo-static Data Flow in GRAPE-II,” Proc. IEEE Wkshp Rapid Sys. Proto., 1994.

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

[8] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang, “Code Optimization Techniques in Embedded DSP Microprocessors,” *DAES*, vol.3, (no.1), Kluwer, Jan. 1998.

[9] P. Marwedel, G. Goossens, editors, *Code Generation for Embedded Processors*, Kluwer, 1995.

[10] S. Moran, I. Newman, Y. Wolfstahl, “Approximation Algorithms for Covering a Graph by Vertex-Disjoint Paths of Maximum Total Weight,” *Networks*, Vol. 20, pp55-64, 1990.

[11] P. K. Murthy, S. S. Bhattacharyya, E. A. Lee, “Minimizing Memory Requirements for Chain-Structured SDF Graphs,” Proc. of ICASSP, Australia, 1994.

[12] P. K. Murthy, S. S. Bhattacharyya, E. A. Lee, “Joint Code and Data Minimization for Synchronous Dataflow Graphs,” *Journal on Formal Methods in System Design*, July 1997.

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

[14] P. K. Murthy, S. S. Bhattacharyya, “Shared Memory Implementations of Synchronous Dataflow Specifications Using Lifetime Analysis Techniques,” Tech report UMIACS-TR-99-32, University of Maryland Institute for Advanced Computer Studies, College Park, MD 20742, http://www.cs.umd.edu/TRs/TRumiacs.html, June 1999.

[15] C. Papadimitriou, K. Steiglitz, *Combinatorial Optimization*, Dover, 1998.

[16] S. Ritz, M. Pankert, and H. Meyr, “Optimum Vectorization of Scalable Synchronous Dataflow Graphs,” Proc. of the Intl. Conf. on ASAP, Oct. 1993.

[17] S. Ritz, M. Willems, H. Meyr, “Scheduling for Optimum Data Memory Compaction in Block Diagram Oriented Software Synthesis,” Proc. ICASSP 95, May 1995.

[18] W. Sung, J. Kim, S. Ha, “Memory Efficient Synthesis from Dataflow Graphs,” ISSS, Hinschu, Taiwan, 1998.

[19] I. Verbauwhede, F. Catthoor, J. Vandewalle, and H. De Man, “In-place Memory Management of Algebraic Algorithms on Application Specific ICs,” *J. VLSI SP*, 1991.

[20] V. Zivojinovic, J. M. Velarde, C. Schlager, H. Meyr, “DSPStone — A DSP-oriented Benchmarking Methodology,” ICSPAT, 1994.

[21] V. Zivojinovic, S. Ritz, H. Meyr, “Retiming of DSP Programs for Optimum Vectorization,” Proceedings of the ICASSP, April, 1994.