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

Praveen K. Murthy
Angeles Design Systems

Shuvra S. Bhattacharyya
University of Maryland

Abstract

Synchronous Dataflow, a subset of dataflow, has proven to be a good match for specifying DSP programs. Because of the limited amount of memory in embedded DSPs, a key problem during software synthesis from SDF specifications is the minimization of the memory used by the target code. We develop a powerful formal technique called buffer merging that attempts to overlay buffers in the SDF graph systematically in order to drastically reduce data buffering requirements. We give a polynomial-time algorithm based on this formalism, and show that code synthesized using this technique results in more than a 60% reduction of the buffering memory consumption compared to existing 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 [15]; 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.

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 [11] from Synopsys, and academic tools like Ptolemy [2] from UC Berkeley, and GRAPE [4]. 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 compiler has a limited view of the code, often confined to basic blocks within each block it is optimizing, and cannot make use of the global control and dataflow that our scheduler can exploit. The techniques we develop in this paper are thus complimentary to the work being done on developing better compilers for DSPs [6][7].

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) [5]. In SDF, each actor produces and consumes a fixed number of tokens, and these numbers are known at compile time. In addition, each edge has a fixed initial number of tokens, called delays.

2 Notation and background

Fig. 1(a) shows a simple SDF graph. Each edge is annotated with the number of tokens produced (consumed) by its source (sink) actor. 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 $prd(e) = cns(e)$ 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 [1].

3 Constructing memory-efficient loop structures

In [1], 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, $2B(2C)$ represents the schedule loop with firing sequence $BCCBCC$. 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.

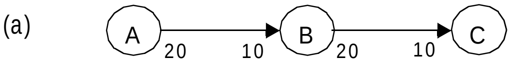
4 Optimizing for buffer memory

Following [1][9], we give priority to code-size minimization over buffer memory minimization. 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 [1] and [9], we also concentrate on acyclic SDF graphs since algorithms for acyclic graphs can be used in the general SAS framework developed in [1].

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.

In [9] and [1], we defined 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 gave 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 developed 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



Valid Schedules

(b)

(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.

buffer will result in significantly less space required than their sums.

5 Merging an input/output buffer pair

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. The theorems are all stated without proof due to lack of space. 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 portions, 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. 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.

5.1 The CBP parameter

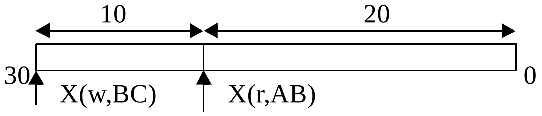


Fig 2. The merged buffer for implementing edges AB and BC in figure 1.

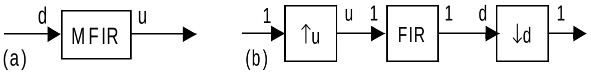


Fig 3. Polyphase FIR filter. The structure in (a) implements the graph (b) efficiently.

We define a parameter called the **consume-before-produce (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 $t \in [0, T]$. 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; we omit the justification for this claim here due to space considerations. Automatic deduction by source code analysis could also be done, but is beyond the scope of this paper. An analysis of optimized assembly language code written for the polyphase FIR filter in the Ptolemy library (figure 3) shows that

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

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 [9], 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 loop factors for iterating these schedules. In other words, the graph can be partitioned into a left subset and a right subset so that the schedule for the graph can be represented as in equation 4. SASs having this form are called R-schedules [9].

Given an R-schedule, we can represent it naturally as a binary tree. The internal nodes of this tree will contain the loop-factor of the subschedule rooted at that node. The leaf nodes will contain the actors, along with their residual loop-factor. Figure 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 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 loop factor 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.

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 a 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.

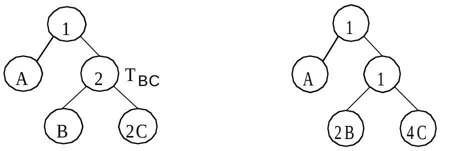


Fig 4. Schedule trees for schedules in figure 1(b)(2) and (3)

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 (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 $T_{XYZ'}$ ($T_{X'YZ}$). I_1 is simply the total number of invocations of Y in the largest subschedule not containing Z (X). 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 $T_{X'Y} \subseteq T_{XYZ'}$ ($T_{YZ'} \subseteq T_{X'YZ}$), where $T_{X'Y}$ ($T_{YZ'}$) is taken to be the largest tree containing Y but not X (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

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 X, 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
$c - p < 0$	$I_1 p + c - p$ $+ \|CBP\|$	$I_1 c + I_2(p - c)$ $+ c - p + \|CBP\|$
$c - p \geq 0$	$I_1 p + I_2(c - p)$ $+ \|CBP\|$	$I_1 c + \|CBP\|$

Note that 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 CBP make it less negative, reducing $|CBP|$.

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

Proof: This can be verified by recognizing that the sum of the buffer sizes when implemented separately is given by $I_1 p + I_2 c$ and $I_1 c + I_2 p$ for the OD and ID cases.

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

Table 2. Merged buffer size for MFIR

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

Observation 2: For homogenous actors like adders, subtractors, and multipliers, (where $c = p$) table 2 simplifies to $I_1 p = I_1 c$ for all cases.

Let $b_i \oplus b_o$ denote the size of the buffer resulting from merging the buffers b_i and b_o, on edges e_i and e_o respectively. 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, $A(b_i \oplus b_o) = b_i \oplus b_o - b_o$. For OD edge pairs, $b_o = I_1 p$ and for ID edge pairs, $b_o = I_2 p$. Hence, table 2, can be rewritten in terms of the augmentation as

Table 3. Augmentation function for MFIR

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

Theorem 2: Let $v_1 \to v_2 \to \ldots \to v_k$ 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,

$$b_1 \oplus \ldots \oplus b_{k-1} = \sum_{i=2}^{k-1} A(b_{i-1} \oplus b_i) + b_{k-1} \quad \textbf{(EQ 5)}$$

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 [8], 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 [8], 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.

6.1 Factoring

In [9], we showed 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. Happily, this result holds for the merging cost function as well, as shown by the following theorem:

Theorem 3: Suppose that $S = (i_L S_L)(i_R S_R)$ is a valid SAS for a chain-structured SDF graph G. Define $\mathrm{cos}t(S)$ to be the size of the buffer obtained by merging all the buffers on the edges in S. Then, the schedule

$$S' = \gamma\left\{\left(\frac{i_L}{\gamma} S_L\right)\left(\frac{i_R}{\gamma} S_R\right)\right\}$$

has $\mathrm{cos}t(S') \leq \mathrm{cos}t(S)$ where γ is an integer that divides i_L, i_R

6.2 DPPO formulation

Let $v_i \to v_{i+1} \to \ldots \to v_j$ 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 $i \leq k < j$ 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 point) 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 [8]. 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. It is a greedy heuristic that attempts to give a good approximation to the minimum. In section 7, we show that on practical SDF systems, this heuristic can give better results than the technique of [8]. In order to compute the interface costs, let the buffers on the edges be $b_i, \ldots, b_k, b_{k+1}, \ldots, b_{j-1}$. Now suppose that the split occurs at k. That is, actors $v_i, \ldots, v_k$ are on the left side of the split. Since we know $\mathrm{cos}t(S_{ik})$ and $\mathrm{cos}t(S_{k+1j})$ (these are memoized, or stored in the dynamic programming table), we have (by theorem 2) that

$$\cos t(S_{ik}) = b_{k-1} + A_{ik}, \text{ and}$$
$$\cos t(S_{k+1j}) = b_{j-1} + A_{k+1j},$$

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:

$$c_{ij}(k) = \begin{matrix} \cos t(S_{ik}) - b_{k-1} + A(b_{k-1} \oplus b_k) \\ + A(b_k \oplus b_{k+1}) + \cos t(S_{k+1j}) \end{matrix} \qquad \textbf{(EQ 6)}$$

We then choose the k that minimizes the above cost:

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

6.3 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. Similarly, define $I_1(v_{k+1}, S_{k+1j})$ for v_{k+1}. Define $I_2(v_k, S_{ij})$ 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^R_{xy} to be the right portion of the schedule S_{xy}, and S^L_{xy} to be the left portion. For instance, if the split in S_{xy} happens at $x \le z < y$, then $S^L_{xy} = S_{xz}$. We can compute I_1, I_2 efficiently, by using the following relationships:

Theorem 4:

$$I_1(v_k, S_{ij}) = q(v_k)/g_{ij}, \; I_1(v_{k+1}, S_{ij}) = q(v_{k+1})/g_{ij},$$
$$I_2(v_k, S_{ij}) = I_2(v_k, S_{ik}),$$
$$I_2(v_{k+1}, S_{ij}) = I_2(v_{k+1}, S_{k+1j})$$

Theorem 5: $I_2(v_k, S_{ik}) = \begin{cases} I_2(v_k, S^R_{ik}), & |S^R_{ik}| \ge 2 \\ q(v_k)/g_{ik}, & |S^R_{ik}| = 1 \end{cases}$,

$$I_2(v_{k+1}, S_{k+1j}) = \begin{cases} I_2(v_{k+1}, S^L_{k+1j}), & |S^L_{k+1j}| \ge 2 \\ q(v_{k+1})/g_{k+1j}, & |S^L_{k+1j}| = 1 \end{cases}$$

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

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 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.

7 Examples

7.1 CD-DAT

Consider the SDF representation of the CD-DAT sample rate conversion example from [8], shown in figure 5.

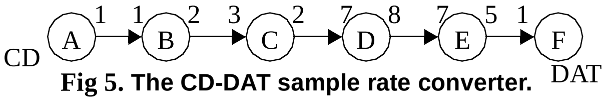


Fig 5. The CD-DAT sample rate converter.

The best schedule obtained for this graph in [8], 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.

7.2 Homogenous SDF graphs

Unlike the techniques in [8][1], 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 6. 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 all current block-diagram code-generators (SPW, Ptolemy,..) 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 2 image buffers would suffice. In order to apply the buffer merging technique to acyclic graphs like this, we need some way of selecting paths along which buffers should be merged (see below). Given a path selection algorithm, our buffer merging technique will easily handle this kind of case, and give an allocation of 3 buffers. In particular, for the example below, we can choose the path to be from A to $Disp$ and apply the merge along that path.

8 Ongoing and future work

As alluded to above, the next step is to extend the buffer merging technique for arbitrary, acyclic SDF graphs, and not just chain-structured graphs. We have developed two algorithms for doing so; one is based on a path-covering algorithm, and the other is based on a bottom-up, clustering approach [10]. However, it is beyond the scope of this paper to discuss these algorithms due to paucity of space. It is also possible to extend our DPPO formulation for arbitrary acyclic graphs (provided we are given a path cover that tells us the sequence of paths to merge buffers on).

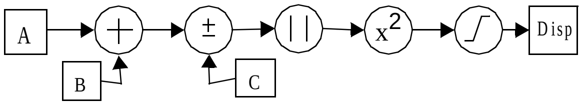


Fig 6. An image processing flow with pixelwise oper tors.

It is also possible to combine the buffer merging technique with other forms of buffer sharing, like lifetime analysis techniques that are based on graph coloring. All of this can be found in [10], along with a complete set of experimental results that shows that the merging technique is indeed very effective for DSP applications. Using retiming to get rid of initial tokens so that a large portion of the graph can be made delayless is a direction for further study.

9 Related work

Our buffer merging work is similar in spirit to the array merging technique presented in [3]; however our technique is significantly less restrictive, and also exploits distinguishing characteristics of SDF schedules in a novel way.

Ritz et. al. [12] 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 [1] that their method yields buffering requirements that are generally even larger than using nested schedules with each buffer implemented separately.

In [13], 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.

The CBP parameter plays a role that is somewhat similar to the array index distances derived in the in-place memory management strategies of Cathedral [14], which applies to nested loop constructs in Silage. However, our 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 Cathedral.

10 Conclusion

In this paper, we have developed a powerful new SDF scheduling technique that improves upon previous efforts demonstrably. This is the technique of buffer merging, a technique that 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 minimizes this merge cost function. The usefulness of the approach has been demonstrated by applying it to a practical system and showing an improvement of more than 20%, and more than 60% on an image-processing example.

11 References

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

[2] 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.

[3] E. De Greef, F. Catthoor, H. De Man, "Array Placement for Storage Size Reduction in Embedded Multimedia Systems," Intl. Conference on Application Specific Systems, Architectures, and Processors, pp. 66-75, July 1997.

[4] R. Lauwereins, P. Wauters, M. Ade, and J. A. Peperstraete, "Geometric parallelism and cyclo-static data flow in GRAPE-II," IEEE Wkshp Rapid Sys. Proto., 1994.

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

[6] 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.

[7] P. Marwedel, G. Goossens, *Code Generation for embedded processors*, Kluwer, 1995.

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

[9] P. K. Murthy, S. S. Bhattacharyya, E. A. Lee, "Joint code and data minimization for synchronous dataflow graphs," *J. on Formal Methods in Sys. Design*, July 1997.

[10] P. K. Murthy, S. S. Bhattacharyya, "Buffer Merging Techniques for Reducing Memory Requirements in SDF Graphs," UMIACS/CS TR, U. Maryland, College Park, MD 20742, http://www.cs.umd.edu/TRs/TRumiacs.html, Oct. 1999.

[11] S. Ritz, M. Pankert, and H. Meyr, "Optimum vectorization of scalable synchronous dataflow graphs," Proc. of the Intl. Conf. on ASAP, Oct. 1993.

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

[13] W. Sung, J. Kim, S. Ha, "Memory efficient synthesis from dataflow graphs," ISSS, Hinschu, Taiwan, 1998.

[14] 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.

[15] V. Zivojinovic, J. M. Velarde, C. Schlager, H. Meyr, "DSP-Stone — A DSP-oriented Benchmarking Methodology," ICSPAT, 1994.