[22] E. A. Lee and D. G. Messerschmitt, “Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing,” *IEEE Transactions on Computers*, January 1987.

[23] E. A. Lee and D. G. Messerschmitt, “Synchronous Dataflow,” *Proceedings of the IEEE*, September 1987.

[24] J. R. McGraw, S. K. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, I. Dobes, P. Hohensee, “SISAL: Streams and Iteration in a Single-Assignment Language”, Language Reference Manual, Version 1.1, July 1983.

[25] P. Papamichalis and R. Simar, Jr., “The TMS320C30 Floating-Point Digital Signal Processor,” *IEEE Micro*, December 1988.

[26] D. B. Powell, E. A. Lee, W. C. Newman, “Direct Synthesis of Optimized DSP Assembly Code From Signal Flow Block Diagrams,” *Proceedings of ICASSP*, San Francisco, California, March 1992.

[27] H. Printz, “Automatic Mapping of Large Signal Processing Systems to a Parallel Machine”, Ph. D. Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213-3890.

[28] S. Ritz, M. Pankert, H. Meyr, “High Level Software Synthesis for Signal Processing Systems”, *Proceedings of the International Conference on Application Specific Array Processors*, Berkeley, CA, August, 1992.

[29] G. Sih, “Multiprocessor Scheduling to Account for Interprocessor Communication,” Ph.D. Thesis, University of California at Berkeley, 1991.

[30] W. W. Wadge, E. A. Ashcroft, “Lucid, the Dataflow Programming Language”, Academic Press, 1985.

[31] M. Yannakakis and F. Gavril, “The Maximum k-colorable Subgraph Problem for Chordal Graphs,” Information Processing Letters 24, 1987.

References

[1] A. V. Aho, R. Sethi, J. D. Ullman, "Compilers Principles Techniques and Tools," Addison-Wesley, 1986.

[2] S. S. Bhattacharyya and E. A. Lee, "Scheduling Synchronous Dataflow Graphs for Efficient Looping," *Journal of VLSI Signal Processing*, to appear in 1992.

[3] J. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, "Multirate Signal Processing in Ptolemy", *ICASSP,* Toronto, Canada, April 1991.

[4] J. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems", *International Journal of Computer Simulation*, June 1992.

[5] M.C. Carlisle, E. L. Lloyd, "On the k-coloring of Intervals", *Advances in Computing and Information — ICCI 1991*, Ottowa, Canada, Lecture Note 497 — Springer Verlag, May 1991.

[6] J. B. Dennis, "First Version of a Data Flow Procedure Language," MIT/LCS/TM-61, Laboratory For Computer Science, MIT, 545 Technology Square, Cambridge MA 02139.

[7] J. B. Dennis, "Stream Data Types for Signal Processing", Technical Report, September, 1992.

[8] J. Fabri, "Automatic Storage Optimization", UMI Research Press, Ann Arbor, Michigan, 1982.

[9] J. L. Gaudiot, L. Bic (editors), "Advanced Topics in Data-Flow Computing," Prentice Hall, 1991.

[10] D. Genin, P. Hilfinger, J. Rabaey, C. Scheers, H. De Man, "DSP Specification Using the Silage Language", ICASSP, Albuquerque, New Mexico, April 1990.

[11] D. Genin, J. De Moortel, D. Desmet, E. Van de Velde, "System Design, Optimization, and Intelligent Code Generation for Standard Digital Signal Processors", *International Symposium on Circuits and Systems*, Portland, Oregon, May 1989.

[12] M.C. Golumbic, "Algorithmic Graph Theory and Perfect Graphs," Academic Press, 1980.

[13] L. J. Hendren, G. R. Gao, E. R. Altman, C. Mukherjee, "A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs", *Lecture Notes in Computer Science*, February 1992.

[14] J. L. Hennessy, D. A. Patterson, "Computer Architecture A Quantitative Approach," Morgan Kauffman Publishers, Inc., 1990.

[15] W. H. Ho, "Code Generation For Digital Signal Processors Using Synchronous Dataflow," *Master's Degree Report*, University of California at Berkeley, May 1988.

[16] W. H. Ho, E. A. Lee, and D. G. Messerschmitt, "High Level Dataflow Programming for Digital Signal Processing," *VLSI Signal Processing III*, IEEE Press, 1988.

[17] G. Kane, "MIPS RISC Architecture", Prentice Hall, 1987.

[18] K. W. Leary, W. Waddington, "DSP/C: A Standard High Level Language for DSP and Numeric Processing", *ICASSP*, Albuquerque, New Mexico, April 3-6, 1990.

[19] E. A. Lee, "Programmable DSP Architectures: Part I," *IEEE ASSP Magazine*, October 1988.

[20] E. A. Lee, "Static Scheduling of Data-Flow Programs for DSP," *Advanced Topics in Data-Flow Computing*, edited by J. L. Gaudiot and L. Bic, Prentice Hall, 1991.

[21] E. A. Lee, W. H. Ho, E. Goei, J. Bier, and S. S. Bhattacharyya, "Gabriel: A Design Environment for DSP," *IEEE Transactions on Acoustics, Speech and Signal Processing*, November 1989.

Schedule: ((2HJKJHK)HJ(3KJH)))

(a)

$\Lambda_1 = (2HJKJHK)$

$in(H^1) = \varnothing$

$in(J^1) = \{H^1\}$ $\quad \phi(H^1,J^1)=T$

$in(K^1) = \{J^1\}$

$in(J^2) = \{J^1,K^1\}$

$\quad \phi(J^1,J^2)=T$

$\quad \phi(K^1,J^2)=T$

$in(H^2) = \{J^2\}$

$in(K^2) = \{J^2,H^2\}$

$J\text{-first}(\Lambda_1) = J^1$

$J\text{-out}(\Lambda_1) = out(K^2) = \{J^2,H^2,K^2\}$

$S_2 = (\Lambda_1 HJ(3KJH))$

$\Lambda_2 = (3KJH)$

$in(K^3) = \varnothing$

$in(J^4) = \{K^3\}$ $\quad \phi(K^3,J^4)=T$

$in(H^4) = \{J^4\}$

$J\text{-first}(\Lambda_2) = J^4$

$J\text{-out}(\Lambda_2) = out(H^4) = \{H^4,J^4\}$

$S_3 = (\Lambda_1 HJ\Lambda_2)$

$\Lambda_3 = (\Lambda_1 HJ\Lambda_2) \Rightarrow (\infty\ \Lambda_1 HJ\Lambda_2)$

$in(\Lambda_1) = \varnothing$ $\quad \phi(J^2,J^1)=T$

$\quad \phi(H^2,J^1)=T$

$\quad \phi(K^2,J^1)=T$

$in(H^3) = \{J^2,H^2,K^2\}$

$in(J^3) = \{J^2,H^2,K^2,H^3\}$

$\quad \phi(J^2,J^3)=T$

$\quad \phi(H^2,J^3)=T$

$\quad \phi(K^2,J^3)=T$

$\quad \phi(H^3,J^3)=T$

$in(\Lambda_2) = \{J^3\}$

$\quad \phi(J^3,J^4)=T$

$\quad \phi(H^4,J^4)=T$

$\quad \phi(J^4,J^4)=T$

$J\text{-first}(\Lambda_3) = J\text{-first}(\Lambda_1) = J^1$

$J\text{-out}(\Lambda_3) = out(\Lambda_2) = \{H^4,J^4\}$

$S_4 = \Lambda_3$

$in(\Lambda_3) = \varnothing$

$\quad \phi(H^4,J^1)=T$

$\quad \phi(J^4,J^1)=T$

(b)

Fig 28. An illustration of how the first-reaches matrix is constructed. For the schedule in part(a), part(b) shows step-by-step how the relevant reachability information is extracted to construct the columns of ϕ associated with actor J.

algorithm iterations until we have collapsed the infinite loop that encapsulates the schedule period.

Figure 28(b) illustrates the construction of the first-reaches table based on the method presented in this appendix.

else

$out(C_i) = in(C_i) \cup \{C_i\}$

if $i \neq r$ then $in(C_{i+1}) = out(C_i)$

These rules are summarized in figure 27(a) and 27(b). Observe that we can describe $in(C_i)$ as the set of CCSS's in Λ_1 which reach C_i before they reach a CCSS for A^*. Thus when we encounter a CCSS C_A associated with A^*, we set each entry in column C_A of ϕ that corresponds to an element of $in(C_A)$.

After processing Λ_1 in this manner, we "collapse" the body of Λ_1 into a single *loop-CCSS* in S. For example, if S = (2A(3 CBA))[1] and Λ_1 represents the loop (3CBA), then we collapse Λ_1 to obtain the hierarchical schedule $S_1 = (2A\Lambda_1)$. We associate two parameters with Λ_1: A^*-first(Λ_1), which denotes the lexically-first CCSS for A^* in Λ_1 (if A^* does not appear in Λ_1 then we write A^*-first$(\Lambda_1) = \varnothing$); and A^*-out(Λ_1), which simply denotes $out(C_r)$. For example, suppose that S is the looped schedule shown in figure 28(a). Let H^1, H^2, H^3 and H^4 denote the CCSS's corresponding to successive appearances of H in the looped schedule, and similarly define CCSS's J^1, J^2, J^3, J^4 and K^1, K^2, K^3 (recall that for SDF actors, subscripts denote invocation numbers, so we use superscripts to label CCSS's). Now suppose that $A^* = J$ and Λ_1 denotes the loop (2HJKJHK). Then A^*-first$(\Lambda_1) = J_1$; A^*-out$(\Lambda_1) = \{H^2, J^2, K^2\}$; and S_2, the loop hierarchy for the next algorithm iteration, is (Λ_1HJ(3KJH)).

In the algorithm iteration corresponding to schedule S_i ($i \geq 2$), we select one of the remaining innermost loops from S_i. This loop Λ_i contains only actor appearances and collapsed loops (members of $\{\Lambda_1, \Lambda_2, \ldots, \Lambda_{i-1}\}$). We process these elements of S_i using the construction rules of figure 27(a) and 27(b) for actor appearances (CCSS's). For each collapsed loop Λ, we apply the rules shown in figure 27(c) instead. Here, rule (2) is required to capture the reachability information associated with successive iterations of Λ, whereas rule (1) corresponds to the flow path entering Λ. After applying the appropriate construction rules to each component of Λ_i, we collapse Λ_i in S_i to obtain S_{i+1}, the schedule for the next algorithm iteration. We proceed through

1. The outermost parenthesis represents the infinite loop that encapsulates the schedule period.

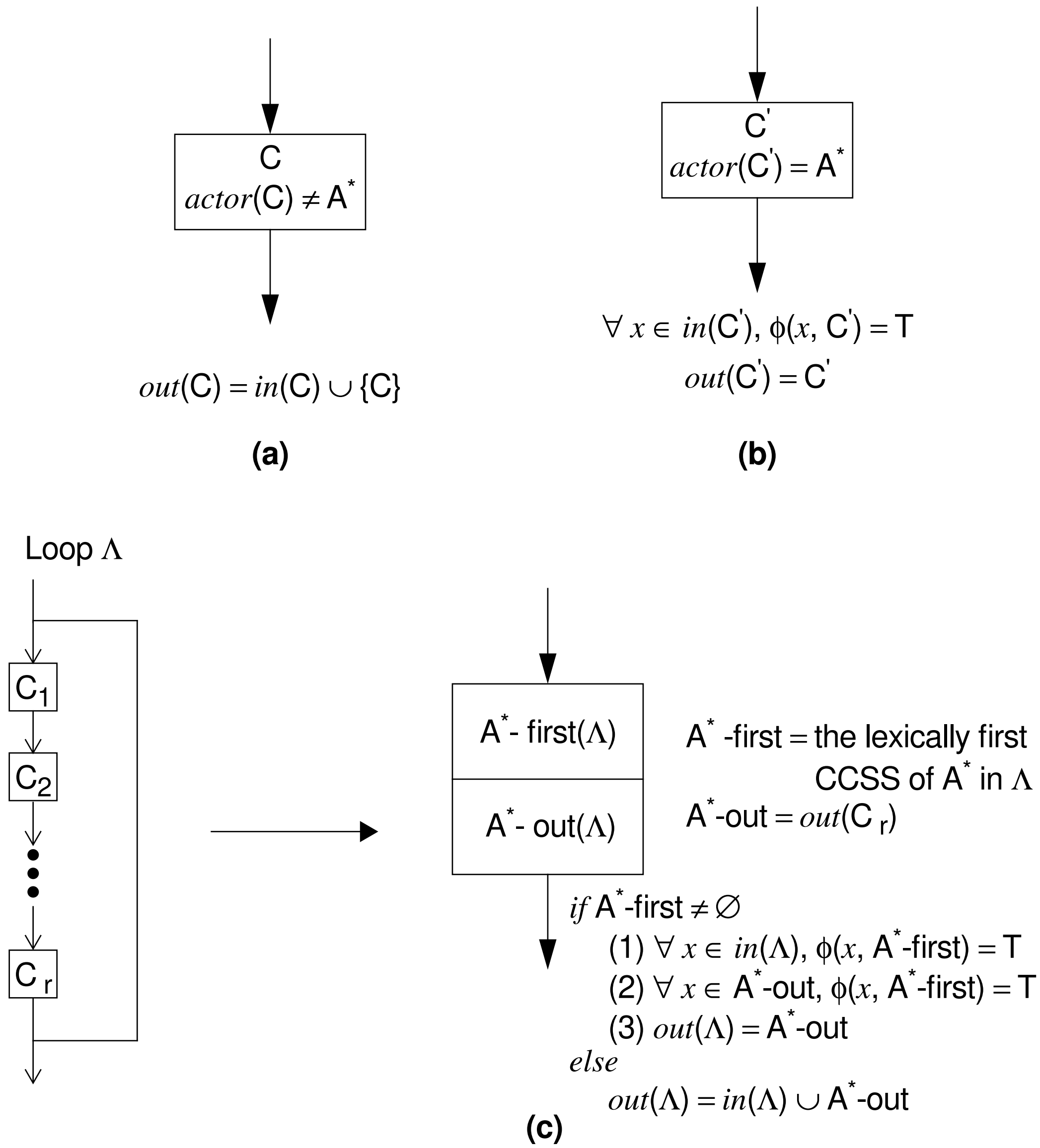


Fig 27. This figure summarizes how the loop structure is hierarchically analyzed to construct the first-reaches table. Parts (a) and (b) correspond to CCSS's in an innermost loop, and part (c) shows how an inner loop is consolidated into a single block C_L in the CCSS flow graph. The pseudocode segment in (c) specifies how C_L is handled when its encapsulating loop is examined. C_1, C_2, …, C_r each represents a CCSS or a consolidated loop.

- $in(C_1) = \varnothing$.
- for $i = 1, 2, \ldots, r$:

  if $actor(C_i) = A^*$, then

    (1) $\forall \eta \in in(C_i)$, set $\phi(\eta, C_i)$ to "T"

    (2) $out(C_i) = \{C_i\}$

been presented. Many of these, such as handling loops, determining buffer parameters, and computing wrap-around accesses will apply frequently, while the importance of various other techniques — e.g. decomposing dynamic buffer components and applying the first-reaches matrix — is very problem-specific. For example, if minimizing chip area is critical, it may be necessary to overlay buffers as much as possible. However, since thorough exploitation of buffer period information is computationally expensive (although not combinatorial), a robust compiler should not attempt it if it is not necessary.

We envision that the large number of specialized optimization strategies introduced in this paper can be best applied within a knowledge-based, goal-oriented framework, such as DESCARTES [27]. We are currently designing such a framework for optimized code generation of multirate signal processing systems. The implementation platform is Ptolemy, an object-oriented prototyping environment for heterogeneous systems [5].

We are also pursuing the incorporation of our memory management strategies into the scheduling process.

9 Appendix

In this appendix, we show how to systematically compute the first-reaches table, which was introduced in section 4. Our technique is an adaptation of the method described in [1] for determining reaching definitions. Let G denote an SDF graph; let S denote a looped schedule for G; let $\phi(\bullet, \bullet)$ denote the corresponding first-reaches table; and recall that for any two CCSS's X and Y associated with G and S, $\phi(X, Y) = T$ if and only if there is a control path from X to Y that does not pass through another CCSS for Y. Also, for any CCSS X, let *actor*(X) denote the actor associated with X — i.e. the actor for which X is a CCSS.

Figure 27 summarizes how to determine the columns of ϕ that correspond to an actor A^* in G. We start by examining some innermost loop Λ_1 of S (by "innermost loop", we mean a loop in which no other loops are nested). Let $C_1, C_2, \ldots, C_r$ denote the CCSS's encapsulated by Λ_1 in lexical order. We process each C_i according to the following construction rules:

delay. This arc represents the address of the current insert-position in the buffer of past samples, which must be maintained from invocation to invocation (for more details, see [15]).

Although a big improvement, this scheme still involves overhead — replicating each new input sample and maintaining the internal circular-buffer pointer. We can eliminate this overhead simply by ensuring that the Nth sample produced on the input arc of the FIR is never overwritten before the (N+n)th sample is consumed. This can be guaranteed by making sure the buffer size is at least n greater than the maximum number of coexisting live samples. Thus, each FIR invocation can read all past samples directly from the buffer associated with the input arc, and no replication is necessary.

This technique applies to any actor which references past samples. The requirement for past samples can be specified by annotating each SDF arc with an additional parameter — the number of past samples required by the sink. Allowing successive invocations to process overlapping "windows" of input samples in this manner also increases the exposure of data parallelism (for details, refer to [27]).

8 Conclusion

Until recently, in the domain of signal processing, graphical programming was primarily used in the context of simulation and developing software for applications with modest performance requirements. When performance requirements approached the limits of the target processor, implementers had to resort to manual programming at the microcode level. However, recent progress in dataflow theory and in compiler technology for dataflow programming now allows compilers for graphical DSP languages to approximate meticulous manual coding for single sample-rate applications.

Although the representation of multirate algorithms as dataflow graphs is well-understood, compiler techniques must be augmented to efficiently manage the iteration and large buffering requirements associated with the multirate case. This paper approaches these problems in a unified manner and develops systematic solutions. A large number of optimization techniques have

actors by the techniques discussed in this section, the compiler should verify compatibility with the loop organization.

Also, if we implement these optimizations by augmenting the language, then we should clearly consider only functions that will be used frequently. However, the ideal solution is to allow the user to define such actors in a programmable fashion. This would allow special-purpose non-computational blocks to be implemented efficiently — for example, an actor that reverses the elements in an array. Supporting this generally requires a significant innovation in the programming model.

In this section, we have presented a class of optimizations for SDF programs based on suppressing the duplication of data by actors that do not perform any computations. There is another widely applicable code optimization that is closely related, and that also requires a deviation from strict SDF semantics — this involves actors whose outputs depend on previously consumed input samples. Probably the most prevalent example of this in DSP applications is the FIR filter. Figure 26 shows one SDF topology for an nth order FIR filter. Here, the homogeneous input arc represents the next sample in the input sequence and the self loop represents the state associated with the FIR block — these are the last n−1 samples of the input sequence. However directly applying this model to compilation would result in $2n$ buffer accesses per invocation! A far preferable solution is to have the code block for the FIR manage the buffering of past samples [15]. This simply involves maintaining a circular buffer of length $n-1$, where each invocation copies the new input sample into the last position, overwriting the oldest buffered sample. In terms of dataflow, this implies replacing the self-loop of figure 26 with a homogenous arc containing unit

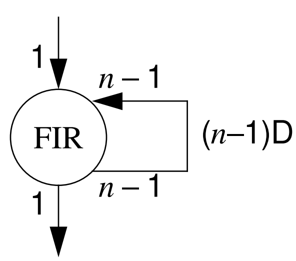


Fig 26. Graphical representation of an nth order FIR filter block.

save four *move* instructions over the conventional implementation of B. If however, we have the looped schedule (3AB)(4C) then it is difficult to apply this optimization. This is because the advances of C_{rp} do not occur in lockstep with the loop L_C that encapsulates C. In particular, if C reads directly from the buffer for A↑B, then C_{rp} advances after the 4th, 8th and 12th read accesses. These correspond respectively to the first access in the second iteration of L_C, the second access in the third iteration of L_C, and the third access in the fourth iteration of L_C. Thus we must test the iteration count after each access to determine whether or not to advance C_{rp}, which will most likely be less efficient than making four physical copies of each input sample to B.

Figure 25 depicts three other common non-computational actors that can be implemented efficiently as "macro buffers" only if the looping structure permits it. Before "optimizing" such

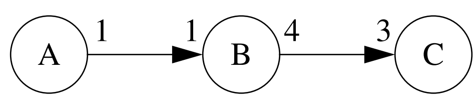


Fig 24. An illustration of how looping can complicate the optimization of non-computational blocks. Here, "B" represents a *repeat* actor that consumes one sample and produces four copies of it on its output arc.

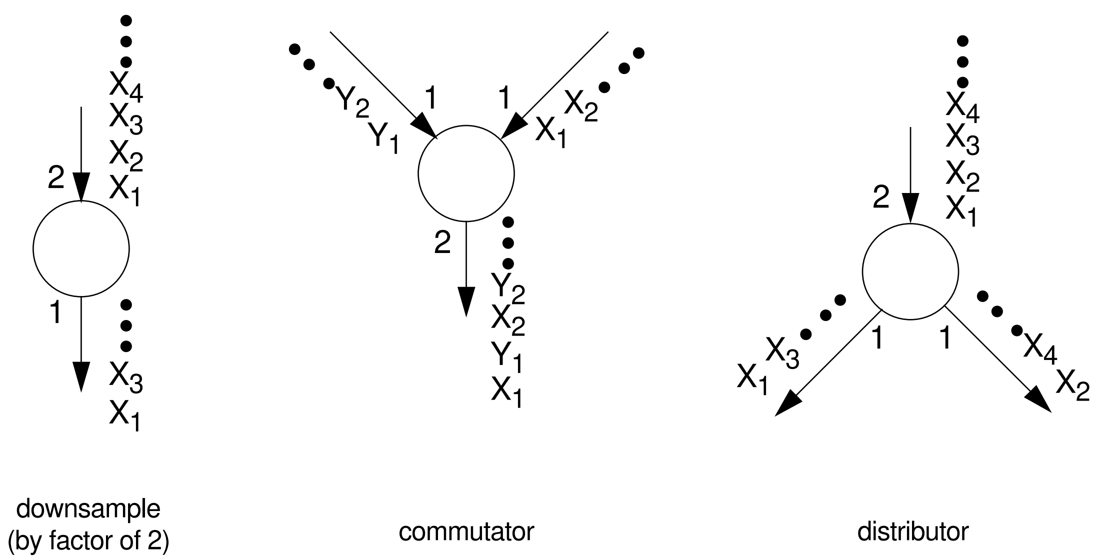


Fig 25. Three useful actors that do not perform operations on their data. "Downsample by factor of n" outputs one out of every n samples consumed. A "commutator" interleaves samples from each input arc onto its output arc; and the "distributor actor" outputs alternate input samples to alternate output arcs.

this scheme. The only additional consideration is that the "macro-buffer" associated with Ψ must be large enough so that a sample is never overwritten before it has been consumed by all destination invocations. The required minimum buffer size is simply the maximum number of coexisting live samples that can exist on any of Ψ's output arcs. The techniques presented in the previous sections of this paper can be extended straightforwardly to the macro-buffers associated with *fork* instances.

We can apply similar optimizations to various other actors that do not perform any operations on their inputs. However, looping often introduces complications. For example, consider the *repeat* actor, which consumes a single sample and replicates this sample n times on its output arc. Figure 24 shows the connection of such an actor (node B, with $n = 4$) to a sink (node C) that consumes three samples per invocation. If there is no looping, we can implement B↑C efficiently by having C's read pointer C_{rp} point directly into the buffer for A↑B and advancing this pointer after every four accesses. Thus no run-time code would be required for the upsample and we would

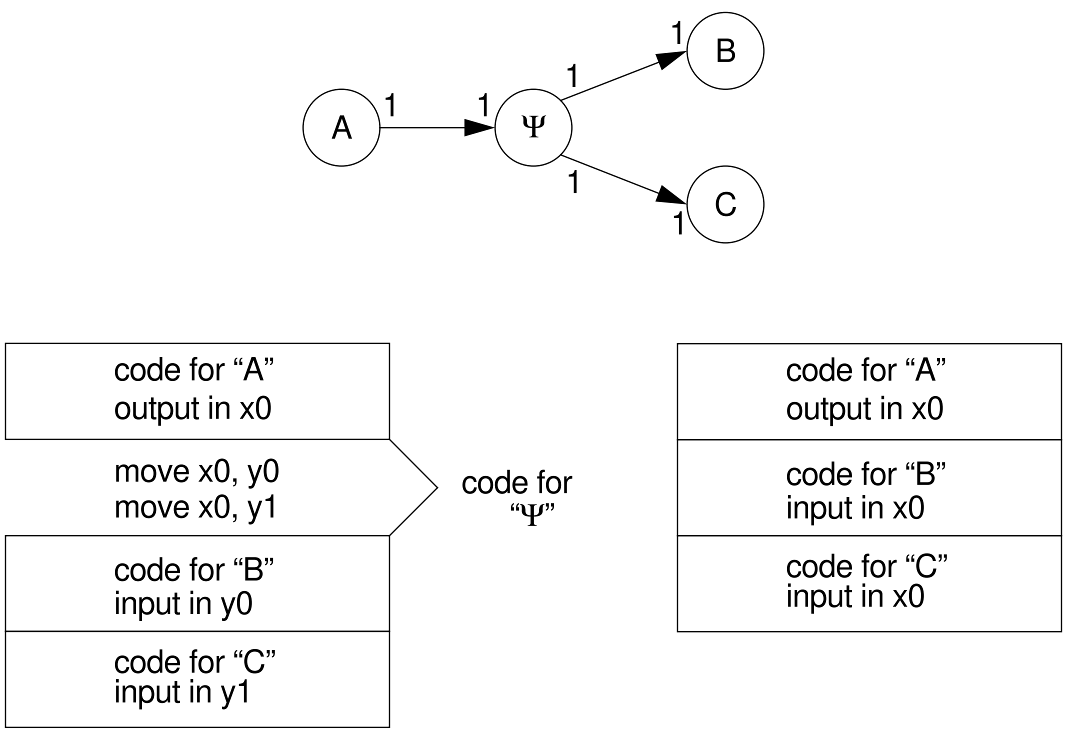


Fig 23. Optimizing the buffering for the *fork* actor.

Λ performs no more than BUFSIZE accesses of b, then we can float the corresponding modulo address computation outside of the loop.

7 Deviating from Dataflow Semantics

The most natural way to compile a dataflow program is to implement each arc as a distinct contiguous block of memory. The production of a sample onto an arc then corresponds to a write into a distinct physical location; and at any given time, there is a one-to-one correspondence between live samples and physical storage locations. We have already shown how modifying this strictly dataflow-based approach to include register allocation and buffer overlaying can improve target code efficiency. For a certain class of actors, a further modification is useful — suppressing the duplication of coexisting samples that have the same value.

Probably the most obvious and most frequently-used example is the *fork* actor, which consumes one input sample and replicates the value of this sample on each of its output arcs. Figure 23 shows a simple illustration of how implementation of *fork* can be optimized. Here, Ψ represents an instance of a 2-output *fork*; A represents an arbitrary homogeneous source actor; and B and C each denote arbitrary homogeneous sinks. The lower left side of the figure shows an outline of Motorola DSP56000 code to implement the graph if Ψ is treated like any other actor (the code outline assumes that register allocation has been performed accross the homogeneous buffer accesses). Since the code associated with Ψ simply copies data, we can eliminate *move* instructions by having B and C read their inputs directly from the output buffer of A. An outline of the resulting code is shown in the lower right side of figure 23. Observe that no extra instructions are required to implement Ψ.

The easiest way to automate this optimization is to make the compiler recognize *fork* as a special actor — we incorporate *fork* into the language. For each instance Ψ of fork, the compiler generates a single logical buffer to implement all of the arcs connected to Ψ. A single write pointer into this buffer is associated with the source of Ψ's input arc, and the sink of each output arc is allocated a distinct read pointer. Thus no run-time code is required to implement the *fork*, except for possible swapping of buffer pointers. Furthermore looping creates no complications for

For a given buffer access instruction sequence, the corresponding machine instructions must perform a modulo address computation if and only the associated access set I_a intersects the set of wrap-around accesses, i.e. iff $I_a \cap Sw \neq \emptyset$. In practice, however we do not need to explicitly compute and maintain S_w nor the access sets associated with each buffer access instruction sequence. We simply simulate the buffer activity, traversing the buffer access instruction sequences in succession, for one schedule period and apply theorem 3 for each access. If Φ denotes the current buffer access instruction sequence in our simulation, and the current access is the *j*th access of arc α by actor A, then we mark Φ as requiring a modulo computation if

$$[\text{delay}(\alpha) + (j - 1)] \bmod \gcd(\text{TNSE}, \text{BUFSIZE}) = \gcd(\text{TNSE}, \text{BUFSIZE}) - 1.$$

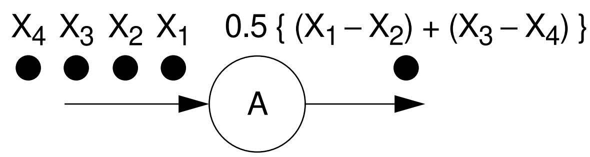
All buffer access instruction sequences which are not marked by this simulation can be translated into simple linear address updates.

6.3 Moving Modulo Address Computations Outside of Loops

Frequently, the wrap-around access for a multirate modulo buffer occurs during the last access associated with each invocation of some schedule loop, allowing us to float the modulo address computation outside of the loop. We illustrated this effect earlier in the example of figure 12. Such situations normally arise through one of two mechanisms. First, when there is no delay on an arc α, optimally looping the firings of source(α) and sink(α) often requires that a loop encapsulate a number of accesses of α equal to the minimum buffer size. The details of this mechanism are beyond the scope of this paper. Second, when there is a delay on α, and the buffer size matches the number of accesses performed by some encapsulating loop Λ_e, then the modulo access associated with sink(α) can be moved outside of Λ_e (this is the case in figure 12). The modulo computation for source(α) must remain inside Λ_e since, due to the delay, the wrap-around access is not the last access of α by source(α) in an invocation of Λ_e.

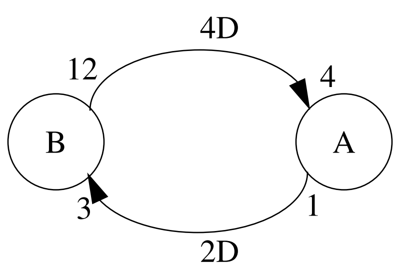
We can detect such opportunities in conjunction with the buffer simulation used to eliminate modulo address computations. For each invocation of a loop Λ, we record the last offset at which this loop invocation accesses each buffer. If at the end of the simulation, we find that each invocation of Λ accesses buffer *b* at the last position (offset BUFSIZE – 1), and each invocation of

space set associated with an instance of this actor will have two buffer access instruction sequences. For instance, for the schedule in figure 22(b), there are four distinct buffer accesses instruction sequences associated with A's connection to B↑A. If we order them lexically, then the first two correspond to the first appearance of A in the schedule, and these represent access sets {1, 3} and {2, 4}; the other two correspond to the second appearance of A, and the associated access sets are {5, 7, 9, 11} and {6, 8, 10, 12}.




```
clear pseudoregister1                          ; initialize the sum
repeat 2                                       ; start of loop
        move input.i++, pseudoregister2        ; consume the next input sample (X1 or X3)
        move input.i++, pseudoregister3        ;   "   "   "   "   "   "  (X2 or X4)
        sub pseudoregister2, pseudoregister3   ; compute the difference
        add pseudoregister3, pseudoregister1   ; update the sum
end-repeat                                     ;
mult 0.5, pseudoregister1                      ; divide the sum by two
move pseudoregister1, output                   ; output the result
```

(a)



Schedule: AB(2A)

(b)

Fig 22. An illustration of distinct *buffer access instruction sequences*.

As we will show in the following subsection, if we assume that the schedule is fixed, we can efficiently eliminate unnecessary modulo address computations using theorem 3 alone — without explicitly computing S_W. However the technique described here for determining S_W should be kept in mind for advanced optimizations which attempt to reorganize the schedule to improve code efficiency. Such techniques might include, for example, selectively unrolling intra-actor loops or schedule loops to isolate modulo address computations. This would require explicit knowledge of each wrap-around access. Incorporating modulo buffer analysis — as well as the other techniques in this paper — into scheduling is an unexplored, but in the authors' estimate, promising area of research.

6.2 Applying the Set of Wrap-Around Accesses

In the absence of looping, the number of modulo computations required in the target code is exactly the number of elements in S_W. However, loops may cause the same physical instructions to perform both wrap-around accesses and linear accesses. In such cases, we must either unroll the loop to isolate the accesses that wrap around, or we must perform a modulo access computation for every access that is executed from within the loop. We do not pursue the issue of unrolling in this paper; it is a topic that our research has not yet addressed. Instead, we focus on analyzing the loop structure to eliminate modulo accesses while leaving the loops intact.

To eliminate unnecessary modulo address computations for the read or write accesses performed by some actor A from/to an arc α, we first identify the set of distinct physical instruction sequences, called *buffer access instruction sequences,* that will be used to access α by A. This is analogous to common code space sets, which associate blocks of program memory with actor invocations. However the buffer access instruction sequences depend on intra-actor loops as well as schedule loops. For example, consider the actor definition in figure 22(a), in which the input arc is accessed through a loop of two iterations. Here "input.i++" specifies the *next* sample in the buffer. Thus the first *move* statement consumes the first and third input samples in successive iterations of the loop, and the second *move* consumes the second and fourth input samples. Each of these *move* statements corresponds to a separate buffer access instruction sequence, since each must be translated to a separate instruction or sequence of instructions. Thus every common code

This check can be further simplified by observing the periodicity of the modulo term above — we need only determine the first wrap-around access j_W explicitly:

$j_W = \mathrm{gcd}(\mathrm{TNSE}, \mathrm{BUFSIZE}) - [\mathrm{delay}(\alpha) \bmod \mathrm{gcd}(\mathrm{TNSE}, \mathrm{BUFSIZE})]$.

Then we immediately obtain the complete set of wrap-around accesses S_W:

$S_W = S_W(\alpha, \mathrm{BUFSIZE}) = \{ j_W + n \times w_S \mid n \in \{0, 1, \ldots, w_S \times \mathrm{floor}[(\mathrm{TNSE} - 1) / w_S]\} \}$,

where $w_S = \mathrm{gcd}(\mathrm{TNSE}, \mathrm{BUFSIZE})$ denotes the window size.

For the example of figure 21 we have $j_W = 5$, and $S_W = \{5, 10, 15\}$. Code to implement these accesses must perform modulo address computations.These modulo computations will correspond to wrap-around accesses only one-third of the time. However, unless we increase the blocking factor, which would in turn increase TNSE, we must ensure that these accesses are always performed with modulo updates. In general, modulo computations will wrap around 1 out of every $n_W = \mathrm{BUFSIZE} / \mathrm{gcd}(\mathrm{TNSE}, \mathrm{BUFSIZE})$ times, and we can reduce the number of modulo computations by a factor of n_W if we increase the blocking factor to n_W. However, the resulting explosion in code space renders this optimization impractical except for extremely simple examples.

Note that the above developments apply to static buffering as well. In this case w_S = gcd(TNSE, BUFSIZE) = BUFSIZE and $n_W = 1$, so each mandatory modulo computation always corresponds to a wrap-around access. Observe also that for both static and dynamic buffers, the number of modulo computations required depends on the choice of the buffer size. Clearly 1 out of gcd(TNSE, BUFSIZE) accesses requires a modulo computation. Thus the modulo overhead varies (neglecting looping considerations) inversely with gcd(TNSE, BUFSIZE). For example in figure 21, a 7-word buffer can support the given schedule. However, this requires 15 / gcd(15, 7) = 15 modulo computations per schedule period — every access must perform a modulo update! Increasing the buffer size to 10 results in 5 times fewer modulo computations. Thus, for critical sections of the code, it may be beneficial to explore tolerable increases in buffer size for the possible reduction of modulo updates, particularly when the cost of the modulo update is high.

Now let w_S denote gcd(TNSE, BUFSIZE), the size of each window. Also let n_W = BUFSIZE / w_S, the number of windows. Suppose that in the first schedule period, access i occurs at offset j of window w (assume now that offsets and windows are numbered starting at 0). Then the window number of the ith access in some later schedule period k can be expressed as $(w + k\mathrm{TNSE}/w_S) \bmod n_W$. This is simply the initial window number plus the number of windows traversed modulo the number of windows. To this expression, we can apply lemma 2 with $p = \mathrm{TNSE}/w_S = \mathrm{TNSE}/\mathrm{gcd}(\mathrm{TNSE}, \mathrm{BUFSIZE})$; $q = n_W = \mathrm{BUFSIZE}/\mathrm{gcd}(\mathrm{TNSE}, \mathrm{BUFSIZE})$; and $r = w$. Interpreting this result, we see that for each window w^*, there will be schedule periods (values of "k") in which the jth access occurs in w^*. Thus the jth access of some schedule period will be a wrap-around access if and only if the jth access of the first schedule period occurs at the end of a window. We have proved the following theorem.

Theorem 3: Suppose α is an SDF arc. Then the jth access ($j \in \{1, 2, \ldots, \mathrm{TNSE}\}$) of source($\alpha$) or sink($\alpha$) is a wrap-around access in some schedule period if and only if

$$[\mathrm{delay}(\alpha) + (j - 1)] \bmod \mathrm{gcd}(\mathrm{TNSE}, \mathrm{BUFSIZE}) = \mathrm{gcd}(\mathrm{TNSE}, \mathrm{BUFSIZE}) - 1.$$

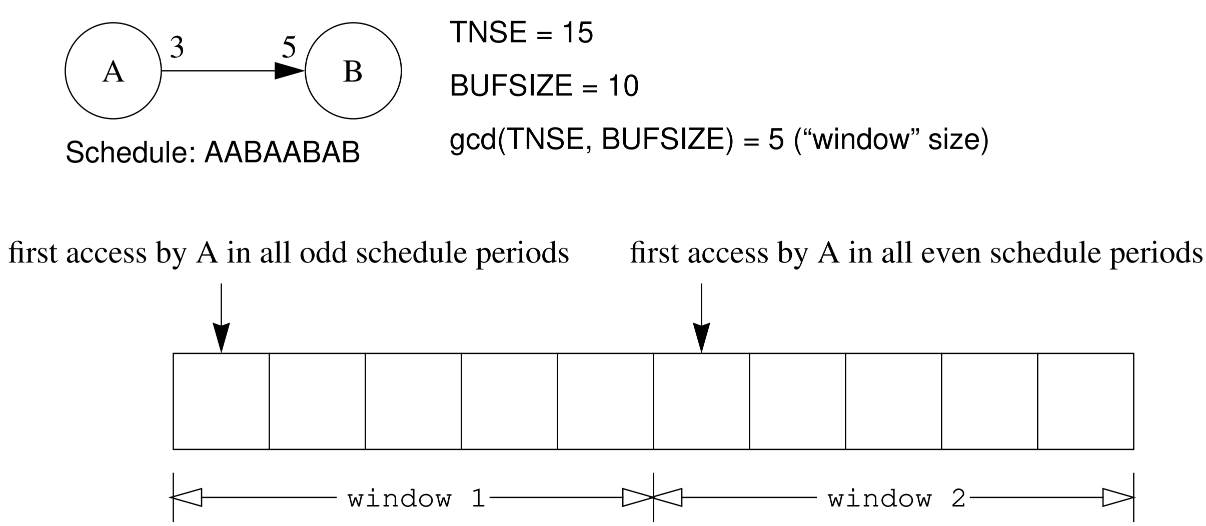


Fig 21. An illustration of repetitive access patterns in gcd(TNSE, BUFSIZE) windows.

Proof: $c \bmod b = c - \text{floor}(c / b) \times b$. Both the subtrahend and minuend of the LHS are divisible by a, so a must divide $c \bmod b$. *QED*

Lemma 2: Suppose that p and q are coprime positive integers, let I_q denote $\{0, 1, \ldots, q-1\}$, and suppose $r \in I_q$. Then $\forall\, k_1 \in I_q \,\exists\, k_2 \in I_q$ such that $(r + pk_2) \bmod q = k_1$.

Proof: Suppose that for some k_1, no such k_2 exists. Then $[(r + px) \bmod q]$ takes on at most $(q-1)$ distinct values as x varies accross I_q. Thus there exist distinct $k_{2a}, k_{2b} \in I_q$ such that

$(r + k_{2a}\, p) \bmod q = (r + k_{2b}\, p) \bmod q = k$, for some $k \in I_q$.

Which implies that there exist distinct nonnegative integers r_a and r_b such that

$r + k_{2a}\, p = r_a\, q + k$, and $r + k_{2b}\, p = r_b\, q + k$.

$\Rightarrow (k_{2a} - k_{2b})\, p = (r_a - r_b)\, q$.

Since p and q are coprime, it follows that $(k_{2a} - k_{2b})$ is a multiple of q. This contradicts our assumption that $k_{2a}, k_{2b} \in \{0, 1, \ldots, q-1\}$. *QED*

Applying lemma 1, with $a = \text{gcd(TNSE, BUFSIZE)}$, $b = k_1\text{TNSE}$, and $c = \text{BUFSIZE}$, we see that

$\forall$ positive integers k_1 $\exists$ a positive integer k_2 such that

$(k_1\text{TNSE mod BUFSIZE}) = k_2\text{gcd(TNSE, BUFSIZE)}$.

This means that we can consider each dynamic buffer as successive "windows" of size gcd(TNSE, BUFSIZE). In some schedule period, if source(α) or sink(α) performs its *i*th access at offset j of window w_x, then, since the *i*th access shifts TNSE positions from schedule period to schedule period, we know that the *i*th access in any schedule period will occur at offset j of some window. For example, for the dynamic buffer in figure 21, it is easy to verify that for all schedule periods, the window offset for A's first access is 0.

6 Eliminating Modulo Address Computations

In section 4 we discussed the overhead associated with accessing circular buffers and we presented examples of how we could reduce this overhead with careful compile-time analysis. We showed that in the absence of looping, we need only perform modulo address-register updates for accesses that wrap around the end of a circular buffer. We also presented examples of how modulo accesses can be eliminated even in the presence of looping. In this section we develop a systematic approach to eliminating modulo accesses.

6.1 Determining Which Accesses Wrap Around

First, we show how to efficiently determine which accesses of a circular buffer wrap around the end of the buffer. For a static circular buffer this is straightforward — we simply determine the values of $n \in [0, \mathrm{TNSE} - 1]$ for which

$$\mathrm{delay}(\alpha) + n = (\text{some positive integer}) \times \mathrm{BUFSIZE},$$

where α denotes the arc in question, and BUFSIZE denotes the length of the circular buffer.

For dynamic buffers, different accesses will wrap around the end of the buffer in different schedule periods. However there may still exist invocations whose accesses do not wrap around in any schedule period. To determine these invocations we need to use a few simple facts of modulo arithmetic.

Lemma 1: Suppose a, b and c are positive integers, and suppose that a divides b and c. Then a divides (c mod b).

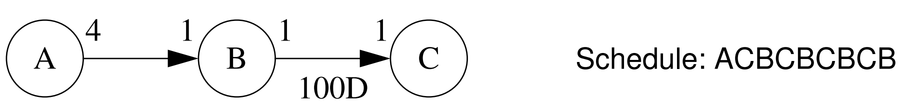


Fig 20. An example showing the benefits of decomposing the dynamic buffer component into a separate segment for each dynamic transaction.

From the above discussion, the number of dynamic samples when $R = M(\alpha)$ is $M(\alpha) - N_s$, and this amount of dynamic samples cannot be exceeded with any other value of R. Therefore $N_d = M(\alpha) - N_s$, which leads immediately to the desired result. *QED*

We conclude this section by pointing out that it is possible to decompose the dynamic buffer component further — each dynamic transaction can be mapped to an independent block. For example, the dynamic buffer component in figure 19 can be separated into three two-word fragments corresponding to transactions (3, 1), (4, 2) and (6, 4). This could be achieved simply by using different read and write pointers for each of the associated accesses — we would need three separate write pointers for A[3], A[4] and A[6] and three separate read pointers for B[1], B[2] and B[4]. The overhead associated with this scheme is significant, but difficult to gauge precisely. First, it places more pressure on the address-register allocator and may increase the amount of spilling. This, in turn requires an extra memory location to save each spilled item. Finally, the sum of the independent dynamic transaction segments (in this case 2 + 2 + 2 = 6) may exceed the maximum number of coexisting dynamic samples (in this case 5). Thus, for small to moderate dynamic buffer sizes it is unlikely that decomposing the dynamic buffer component further will be of value. However, when large delays are involved, it may provide substantial new opportunities for overlaying. For example, in figure 20 there are no static transactions for $B\uparrow C$, and a 100 word block of memory is required for this arc if we do not decompose the dynamic buffer component. However, if we view each of the four dynamic transactions ((1, 1), (2, 2) (3, 3) (4, 4)) as a separate unit, we can implement this arc with four independent 25 word blocks of memory. This additional freedom may lead to much better overall memory use if this example is a subsystem in a more complex graph.

Since currently we cannot effectively predict the tradeoffs in decomposing the dynamic buffer component, we have no systematic procedure for determining precisely when the optimization is useful. This is a topic for further research.

words. Although the net requirement of physical memory is the same (8 words), there is less potential for fragmentation, or equivalently, more opportunity for buffer reuse [8] when this example is a subsystem in a larger graph. Furthermore, the lifetime of β extends through the entire schedule period, whereas L2 and L3 are live only in the interval between A1 and B2. These two locations may thus be reused for other parts of the graph.

It is not obvious however, that decomposing a buffer based on static and dynamic transactions will never increase the net memory requirements. If we refer to the samples associated with static transactions and dynamic transactions as *static samples* and *dynamic samples* respectively, then the transaction-based decomposition requires a set of blocks whose sizes total $N_s + N_d$ words, where N_s is the number of static samples (in a single schedule period) and N_d is the maximum number of coexisting dynamic samples. If this sum exceeds the maximum number of coexisting samples on the arc, then without further analysis — for which currently there are no general techniques — we cannot guarantee that decomposing the buffer will not be detrimental. Fortunately, however, $(N_s + N_d)$ is always equal to the undecomposed dynamic buffer size, as the following theorem proves.

Theorem 2: Suppose that α is an SDF arc for which the maximum number of coexisting samples $M(\alpha)$ exceeds TNSE. Then $N_s + N_d = M(\alpha)$.

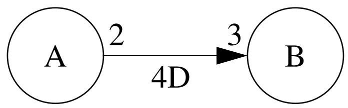
Proof: Suppose at some time τ in the schedule period there are R live samples on α, and first suppose that $R \geq TNSE$. Since the tokens buffered on an arc are successive, the last TNSE samples produced by source(α) are on the arc. Thus, there is a sample corresponding to each static transaction on the arc. It follows that there are $R - N_s$ dynamic samples on α at time τ. Now suppose that $R < TNSE$. We consider two cases here:

Case 1: $(R < TNSE)$ and $(N_s < TNSE - R)$. Then

(The number of dynamic samples at time τ) $\leq R \leq TNSE - N_s < M(\alpha) - N_s$.

Case 2: $R < TNSE)$ and $(N_s \geq TNSE - R)$. Then

(The number of dynamic samples at time τ) $\leq R - [N_s - (TNSE - R)] = TNSE - N_s < M(\alpha) - N_s$



Schedule: AABAB

transactions

(1, 5) <static>
(2, 6) <static>
(3, 1) <dynamic>
(4, 2) <dynamic>
(5, 3) <static>
(6, 4) <dynamic>

read access	number of samples on $A \uparrow B$ just prior to the access
B[1]	8
B[2]	7
B[3]	6
B[4]	7
B[5]	6
B[6]	5

(a)

	L1	L2	L3	Dynamic Buffer Component				
initially	1,3			1,1	1,2	1,4		
after A_1	1,3	1,5	1,6	1,1	1,2	1,4		
after A_2	1,3	1,5	1,6	1,1	1,2	1,4	2,1	2,2
after B_1		1,5	1,6			1,4	2,1	2,2
after A_3	2,3	1,5	1,6	2,4		1,4	2,1	2,2
after B_2	2,3			2,4			2,1	2,2

(b)

Fig 19. An illustration of static transactions and dynamic transactions for a dynamic buffer. In (b), "i,j" represents the live sample which is to be the jth sample consumed by B in schedule period i.

The transactions on an arc can be determined easily from the acyclic precedence expansion graph, or APEG (see [29] for a systematic procedure for computing the APEG associated with an SDF graph), and the static and dynamic transactions can be identified by simulating the activity on the arc over one schedule period. Figure 19 illustrates the decomposition of a buffer based on static and dynamic transactions. In this example, TNSE is 6 while the maximum number of coexisting samples on $A \uparrow B$ is 8 — so clearly dynamic buffering applies. However, from the table on the right side of figure 19(a), we see that the third, fifth and sixth read accesses of B occur when there are TNSE or fewer samples queued on $A \uparrow B$. This corresponds to the set of static transactions, which is summarized in the table on the left side of figure 19(a). Thus samples associated with transactions (1, 5), (2, 6) and (5, 3) can be buffered in independent memory locations, while (3, 1), (4, 2) and (6, 4) must be maintained in a single contiguous block of memory. The resulting constraint sets are {A[1]}, {A[2]}, {A[5]}, {A[3], A[4], A[6]}. Figure 19(b) illustrates the use of these constraint sets to form independent buffering units. Here, A[1], A[2] and A[5] are mapped to independent (not necessarily contiguous) memory locations L1, L2 and L3 respectively, and the remaining constraint set is mapped to a five-word contiguous block of storage, labeled the "dynamic buffer component". Five words are required because this is the maximum number of coexisting live samples from {A[3], A[4], A[6]}. Figure 19(b) shows how the profile of live samples in this buffering arrangement changes through the first schedule period. Each live sample is represented by an ordered pair *i,j*, which denotes the *j*th sample to be consumed by B in schedule period *i*, and a shaded region designates the absence of a sample. Observe that for each live sample *s* in the dynamic buffer component, there is some point in the schedule period when *s* coexists with the corresponding sample of the next or previous period. This is precisely why these samples must be buffered as a contiguous unit. Observe also that in the dynamic buffer component, the read and write pointers for B and A, respectively, each shift three positions to the right (in a modulo-5 sense) every schedule period. These pointers are not involved in accesses of L1, L2 and L3 — these locations can be accessed using absolute addressing.

For the example in figure 19, mapping all accesses of $A \uparrow B$ to a single contiguous segment β of memory requires an 8 word block of memory, while decomposing this buffer based on static and dynamic transactions allows a partition into four mutually independent blocks of 1, 1, 1 and 5

5.3 Constraints for Dynamic Buffering

Dynamic buffering imposes contiguity constraints between buffer accesses whenever a read occurs when the number of samples on the arc exceeds TNSE. In such situations, the sample to be read co-exists with the corresponding sample of the next schedule period — so we cannot dedicate a single memory location to that sample. For a given arc, an efficient way do deal with such cases is to force all of these accesses to occur in the same contiguous block β of memory. Since each of these sample's location will vary between schedule periods, they access β through read/write pointers. Any read which occurs when the sample population is within TNSE however, corresponds to a sample whose location is independent of β. To explain this effect precisely, we introduce the following definition:

Definition 3: Let G be an SDF graph and suppose that A↑B is an arc in G.Then a **transaction** on A↑B is an ordered pair (i, j), $1 \le i,j \le \mathrm{TNSE}$, such that[1]

$$j = ([i - 1 + \mathrm{delay}(\mathrm{A} \uparrow \mathrm{B})] \bmod \mathrm{TNSE}) + 1$$

Thus (i, j) is a transaction on A↑B if the jth sample consumed by B in any given schedule period is the ith sample produced by A in that schedule period or some earlier schedule period. For a given periodic schedule S for G, we say that (i,j) is a **static transaction** if the number of samples existing on A↑B just prior to the jth read of B is less than or equal to TNSE. We can express this condition as

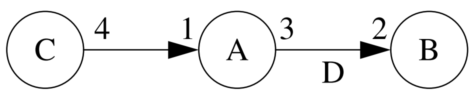
$$[\mathrm{delay}(\mathrm{A} \uparrow \mathrm{B}) + p(\mathrm{A} \uparrow \mathrm{B}) N_A] - [c(\mathrm{A} \uparrow \mathrm{B})(N_B - 1) + (j-1) \bmod c(\mathrm{A} \uparrow \mathrm{B})] \le TNSE,$$

where $N_B = 1 + \mathrm{floor}[(j - 1) / c(\mathrm{A} \uparrow \mathrm{B})]$ is the invocation of B in which the jth read access of A↑B occurs and N_A is the number of invocations of A which precede B_{N_B} in S. Finally, we say that a transaction is a **dynamic transaction** if it is not a static transaction.

1. The "+1" and "-1" are required in this expression because we (by convention) number samples starting at 1 rather than 0.

merge buffer periods {A[7-9]} and {A[10-12]}, and the resulting buffer periods are {A[1-6]} and {A[7-12]}. Finally, if we impose the condition that B reads A↑B through an intra-actor loop, then we have the six additional constraint sets shown in the fifth row of figure 18(b).The first of these constraint sets intersects both of the remaining buffer periods and we are left with a single buffer period {A[1-12]}.

So far we have only mentioned that dynamic buffering can also lead to constraint sets, however we have not described this effect. The effects of dynamic buffering are more subtle than conditions imposed by loops. This is the topic of the next subsection.



Schedule: CAABABBABBB

(a)

Some Possible Constraint Sets

Singletons	{A[1]} {A[2]} … {A[12]}
A writes to A↑B through a loop	{A[1-3]} {A[4-6]} {A[7-9]} {A[10-12]}
Encapsulate A_1, A_2 in a schedule loop	{A[1-6]}
Encapsulate B_5, B_6 in a schedule loop	{A[8-11]}
B reads from A↑B through a loop	{A[12],A[1]} {A[2],A[3]} {A[4],A[5]}
	{A[6],A[7]} {A[8],A[9]} {A[10],A[11]}

(b)

Fig 18. This example illustrates how superimposing different constraint sets can lead to different buffer periods. The figure depicts a multirate graph, a schedule for the graph and five possible constraint sets for the arc A↑B. We use A[*i-j*] as shorthand notation for A[*i*], A[*i*+1], …, A[*j*], if $i < j$.

Definition 2: Given an SDF graph G, an arc α in G, and a schedule S for G, let $C = \{C_1, C_2, \ldots, C_k\}$ denote the collection of constraint sets imposed on α. Suppose $b = \{b_1, b_2, \ldots, b_n\} \subseteq C$ such that

(1) No member of b is independent of all other members of b — if n > 1, then for each b_i there is at least one $b_j \neq b_i$ such that $b_j \cap b_i \neq \varnothing$; and

(2) b is independent of the remainder of C — i.e. $\left(\bigcup_{z=1}^{n} b_z\right) \cap \left[\left(\bigcup_{z=1}^{k} C_z\right) - \left(\bigcup_{z=1}^{n} b_z\right)\right] = \varnothing$

Then $\left(\bigcup_{z=1}^{n} b_z\right)$ is called a **buffer period** for α.

One can easily verify that for a given schedule, each arc has a unique partition into buffer periods. Furthermore, samples in the same buffer period must be mapped to the same contiguous physical buffer whereas distinct buffer periods can be mapped to different segments of memory. Finally, the amount of memory required for a buffer period is simply the maximum number of coexisting live samples in that buffer period. Figure 18(a) depicts an example which we will use to illustrate the consolidation of different constraint sets into buffer periods. The schedule of figure 18(a) does not contain any loops. If the buffer accesses within A or B do not occur within intra-actor loops, then only the singleton constraint sets apply to $A\uparrow B$, and the buffer periods are {A[1]}, {A[2]}, …, {A[12]}.

Now suppose A accesses $A\uparrow B$ through a loop inside A. The corresponding constraint set is shown in the second row of figure 18(b), and we obtain the resulting buffer periods by superimposing the first two rows of figure 18(b) — {A[1-3]}, {A[4-6]}, {A[7-9]}, {A[10-12]}. If we add the additional condition that the first two invocations of A are grouped into a schedule loop (we change the schedule to C(2A)BABBABBB), then we must consider another constraint set {A[1-6]}. The new buffer periods are the combination of the 17 constraint sets in the first three rows of figure 18(b) — {A[1-6]}, {A[7-9]}, {A[10-12]}. Now if we encapsulate B_5 and B_6 within a schedule loop (the new schedule is C(2A)BABBAB(2B)), the resulting constraint set is {B[9-12]}, which is equivalent to {A[8-11]} due to the unit delay. This new constraint forces us to

Determining the constraints due to schedule loops is also straightforward. Given an arc A↑B and an $X \in \{A,B\}$, each outermost loop L in the periodic schedule defines a constraint set that consists of all accesses by X of A↑B which occur within L. We can derive these from the contiguous ranges of invocations of A and B that L encapsulates. We map all accesses within a loop to the same physical block of memory because we cannot easily perform isolated resets of read/write pointers inside loops. Expensive schemes — such as testing the loop index to determine which physical buffer to use or maintaining an array of buffer locations — are required to fragment buffering within a loop. We do not consider such schemes presently because we expect that their benefits are rare, and thus we consolidate accesses within loops to the same physical buffers.

Note that unlike the constraint sets corresponding to loops inside actors, a constraint set corresponding to a schedule loop does not necessarily require a separate word for each member of the set. A simple example is shown in figure 17. Here, the loop imposes the constraint set {B[1], B[2], … B[20]} for B↑C, but clearly only two words are required to implement the buffer for this arc. The actual memory requirement for each section of a fragmented buffer can easily be determined by simulating the buffer activity over a single schedule period and noting the maximum number of coexisting samples.

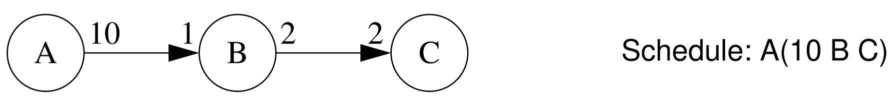


Fig 17. An illustration of compact buffering within a constraint set.

The constraint sets due to intra-actor looping, inter-actor looping and dynamic buffering together define the physically independent sections of a buffer, which we have termed the “buffer periods”. We also include the singleton constraints {A[1]}, {A[2]}, …, {A[TNSE]}, which we need to account for samples that don’t appear in any of the other constraint sets. For an SDF arc α, we refer to the entire collection of constraint sets, including the singleton constraints, as the *collection of constraint sets imposed on* α. Then, determining the buffer periods, which can be viewed as the maximal independent constraint sets, amounts to partitioning the entire collection into maximal nonintersecting subsets.

defined a buffer period to be an indivisible subset of invocations whose accesses of a particular arc are independent of the other invocations that access the arc. This independence allows different buffer periods for the same arc to be mapped to different blocks of memory and it provides an efficient way to fragment the aggregate buffer's lifetime.

There are four mechanisms that can impose contiguity constraints on successive buffer accesses of an arc α — writes to α occurring from a loop inside source(α); reads from α occurring from inside a loop in sink(α); placement of source(α) or sink(α) within a schedule loop; and dynamic buffering. The constraints imposed by these mechanisms can be specified as subsets of samples which must be buffered in the same block of storage. For example, suppose that for the SDF graph in figure 16(a), actor A is programmed so that it writes its samples iteratively. The resulting contiguity constraints are illustrated in figure 16(b) — the three samples produced by each invocation must be stored in three adjacent memory locations. We specify these two con-

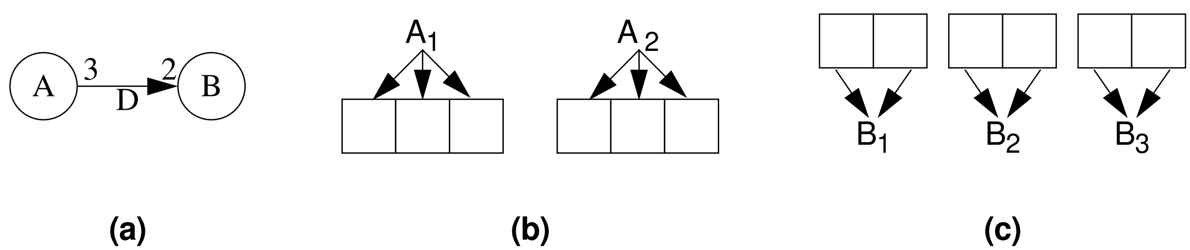


Fig 16. An illustration of buffering constraints when arcs are accessed through loops inside the actors.

straints by the subsets {A[1], A[2], A[3]} and {A[4], A[5], A[6]}, where A[*i*] represents the *i*th sample accessed by A in a schedule period [1] (for $1 \le i \le$ TNSE). The constraints resulting from B's reads occurring from within a loop are depicted in figure 16(c), and we can represent these constraints analogously as {B[1], B[2]}, {B[3], B[4]} and {B[5], B[6]}. However, since we must ultimately superimpose all constraints, we would like to express them in terms of the same actor. Our convention will be to express all contiguity constraints in terms of the source actor. Thus, noting the unit delay on $A \uparrow B$, we translate figure 16(c) to {A[6], A[1]}, {A[2], A[3]}, {A[4], A[5]}.

1. This notation assumes that the arc in question (in this case $A \uparrow B$) is understood.

D_4, B_2} and the first buffer period wraps around the end of the periodic schedule — the first four samples consumed by B in a schedule period are the last four samples produced by D in the previous schedule period. Notice also that each of the buffer periods for B↑C, C↑D and D↑B requires four words of storage. From the lifetime profile, we see that B↑C<1> overlaps with C↑D<1>, C↑D<1> overlaps with D↑B<2>, and D↑B<2> overlaps with B↑C<2>. Thus if we are constrained to map all buffer periods of a given arc to the same block of memory, then three separate 4-word blocks are required for B↑C, C↑D and D↑B. If, however, we consider each buffer period as an independent unit, then from the two lower sections of the lifetime profile, we see that only two 4-word segments suffice — one for {B↑C<1>, D↑B<2>, C↑D<2>} and another for {D↑B<1>, C↑D<1>, B↑C<2>}. Taking into account the 2 words required for A↑B, we see that the decomposition into buffer periods reduces the total storage requirements from 14 words to 10.

5.2 Determining Buffer Periods

In the previous subsection, we illustrated the use of buffer periods to reduce lifetimes and to increase flexibility in allocating memory for contiguous buffers. Now we examine how to systematically determine the buffer periods and to apply them to memory allocation. We have loosely

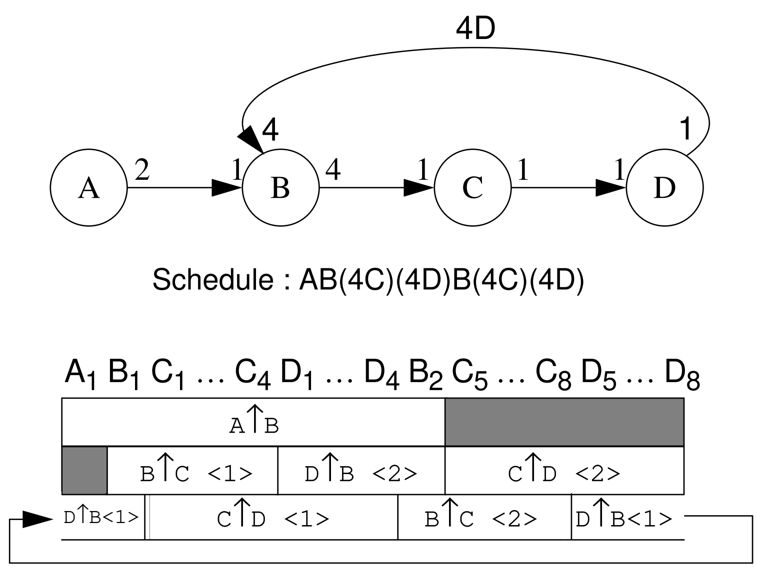


Fig 15. An example of how mapping different buffer periods of an arc to different blocks of memory can improve memory allocation.

suppress the “<1>” index for arcs that have only one buffer period). This new profile reveals that we can map both B↑D and C↑E to the same 10-unit block of storage, because even though the lifetimes of these arcs conflict, the buffer periods do not. Thus the memory requirements can be reduced almost in half to 14 words.

In this example, we have exploited only the reduction in overall lifetime for a decomposition into buffer periods. It is also possible to map different buffer periods for the same arc to different blocks of memory. This technique may be useful for overlaying buffers along multirate cyclic paths in the SDF graph. Consider, for example figure 15. This figure shows an upsampled multirate feedback loop along with the resulting buffer period profiles. Notice that due to the delay of four on D↑B, the buffer periods of this arc are $\{D_5, D_6, D_7, D_8, B_1\}$ and $\{D_1, D_2, D_3,$

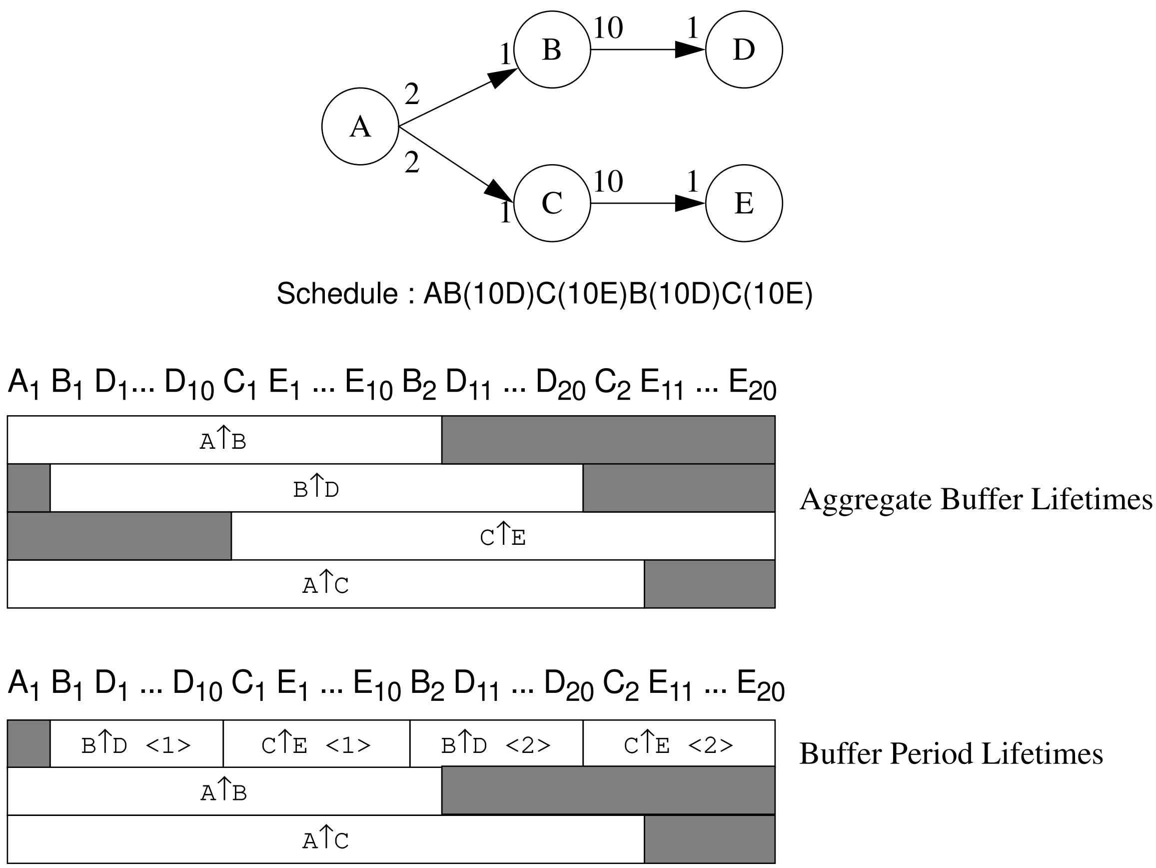


Fig 14. An illustration of opportunities to overlay buffers based on the periodicity of accesses.

contiguous buffer to a separate block of physical memory may require more data-memory space than what is available. In this section, we show how to analyze dataflow properties and properties of the schedule to efficiently determine opportunities for mapping multiple noninterfering buffers to the same physical memory locations. We also show how to determine how to fragment contiguous buffers in physical memory, which can expose more opportunities for overlaying [8]. This precise lifetime and fragmentation information can be used to improve simple first-fit or best-fit storage optimization schemes, which are frequently applied to memory allocation for variable-sized data items. Fabri [8] has studied more elaborate storage optimization schemes that incorporate a generalized interference graph. These schemes are equally compatible with the methods developed in this section.

5.1 Buffer Periods

The periodic nature of buffer accesses can be exploited to fragment an arc's storage into multiple independent contiguous blocks whose combined lifetime does not exceed — and is often much less than — the lifetime of the entire arc. Figure 14 illustrates this effect. Here, a multirate graph is depicted along with a looped schedule for the graph and the resulting buffer lifetime profiles. The first profile treats each arc as an indivisible unit with respect to buffering. In this model, a delayless buffer is assumed to be "live" from the first firing of the source actor until the last firing of the sink actor and for an arc with nonzero delay, a buffer is deemed live throughout the entire schedule period. In the example of figure 14, we see that this straightforward designation of buffer lifetimes does not reveal any opportunity for buffers to share storage and thus $A\uparrow B$, $A\uparrow C$, $B\uparrow D$ and $C\uparrow E$ require 2, 2, 10 and 10 units of storage respectively, for a total of 24 units.

Notice, however, that invocations that access $B\uparrow D$ can be divided into two sets $\{B_1, D_1, D_2, \ldots, D_{10}\}$ and $\{B_2, D_{11}, D_{12}, \ldots, D_{20}\}$ such that all samples are produced in the same set that they are consumed — there is no interaction among the two sets. Thus they can be considered as independent units for storage allocation, with lifetimes ranging from B_1 through D_{10} and B_2 through D_{20} respectively. We call these two invocation subsets the *buffer periods* of $B\uparrow D$, and we denote them by successive indices as $B\uparrow D\langle 1\rangle$ and $B\uparrow D\langle 2\rangle$. The live range for $C\uparrow E$ can be decomposed similarly and the resulting lifetime profile is depicted at the bottom of figure 14 (we

4.5 Summary

Fig 13 illustrates the relationships between the different buffer classifications which we have presented. Any vertical path represents a set of buffer qualities that can coexist. There are four possible combinations — contiguous/static/linear, contiguous/static/modulo, contiguous/dynamic/modulo and scattered/static/linear. This section has provided a systematic approach to determining the qualities of a buffer based on information in the dataflow graph.

We conclude this section with a summary of the situations in which register indirect addressing is desirable:

- The buffer is dynamic.
- The buffer is accessed from within a schedule loop and all members of the CCSS do not access the buffer at the same offset.
- The buffer is accessed from a loop inside the actor.

5 Overlaying Buffers

Recall that storage optimization for the scattered buffers in an SDF program can be formulated in terms of coloring a circular-arc graph and that effective heuristics have been developed for this class of coloring problems [13]. Contiguous buffers do not lend themselves to this technique since their sizes vary [12]. When large sample rate changes are involved, assigning each

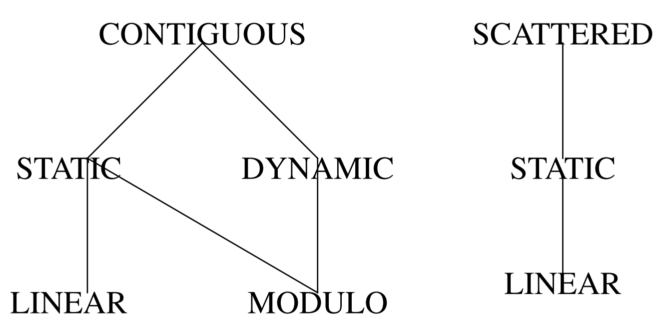


Fig 13. The relationship between the different categories of buffers for an SDF program. Any vertical path represents a set of buffer qualities that can coexist.

resetting C_{rp} to point to the beginning of the buffer *just prior to entering the loop* (2B(2C)). Also, due to the delay on B↑C, the write-pointer for B wraps around after the *first* write access of invocation B_2 in each schedule period. Thus, if B's writes do not occur inside a loop within B, then we can omit the modulo address computation for the first write in the code block for B.

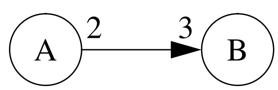
In section 6, we will present general techniques for eliminating modulo accesses. Presently, we conclude that circular buffering may potentially introduce execution-time overhead. For arcs with delay, this risk in unavoidable — circular buffers are mandatory. However, for some delay-free arcs it may be preferable to forego the data-memory savings offered by modulo buffering so that the overhead can be avoided. A buffer size of TNSE clearly guarantees that no modulo accesses will be required — provided that we reset the buffer pointer at the start of every schedule period. Smaller buffer sizes (divisors of TNSE which meet or exceed the maximum number of coexisting samples) are also possible, but one must verify that no access within a loop wraps around the buffer. This expensive check is very rarely worth the effort. A simple rule of thumb can be used for deciding whether to switch to linear buffering for a delayless arc — we prioritize each delayless arc α by the following "urgency measure" μ:

$$\mu(\alpha) = \left\lceil \frac{TNSE(\alpha)}{\text{minimum buffer size of }\alpha} \right\rceil \times \left\lceil \frac{1}{TNSE(\alpha) - (\text{minimum buffer size of }\alpha)} \right\rceil$$

The first bracketed term is the number of modulo accesses that occur on each end of α every schedule period, and the denominator in the second term is the storage cost to convert this arc to a static buffer of size TNSE. Thus, $\mu(\alpha)$ denotes the number of modulo accesses eliminated per word of additional storage. We simply convert the arcs with the highest μ values until we have exhausted the remaining data memory. Many variations on this scheme are possible, and architectural restrictions on the layout of storage, such as multiple independent memories [19], may require modification.

Unfortunately, the presence of loops often precludes the application of this technique. For example, if actor B was programmed with a loop surrounding the input buffer accesses, then the modulo computation would have to be performed in every iteration of the loop even though wrap-around only occurs during the second iteration. A naive policy to account for loops would be to perform a modulo update every time a circular buffer is accessed from within a loop. However, this would prevent us from exploiting an opportunity for optimization which occurs in many multirate graphs.

Figure 12 shows a simple example. Here, due to the unit delay, a circular buffer is required to implement arc $B\uparrow C$. Since TNSE($B\uparrow C$) = 4, a buffer of length four suffices for static buffering. Let C_{rp} denote the readpointer associated with C's accesses of $B\uparrow C$, and observe that C_{rp} wraps around its associated buffer after every fourth access. Since C performs four read accesses throughout each invocation of the loop (2B(2C)), modulo address computation can be avoided by



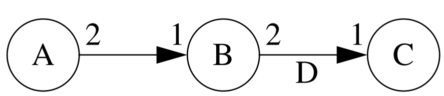
Schedule: AABAB

(a)

```
move X:(r0)+, x0        ; Consume the first input token.
move #44, r0            ; Reset the input buffer's pointer to the start of the buffer.
move X:(r0)+, x1        ; Consume the second input token.
move X:(r0)+, y0        ; Consume the third input token.
............            ; Process the input samples......
............            ;
```

(b)

Fig 11. An example of how modulo address updates can be avoided for circular buffers. Part (a) shows an SDF graph and a schedule for this graph, and part (b) shows sample Motorola 56000 code to implement the buffer accesses of invocation B_2 assuming a buffer size of four.



Schedule A(2B(2C))

Fig 12. An example of an opportunity to optimize modulo buffer accesses within a loop.

4.4 Linear vs. Modulo Buffering

For each contiguous buffer, we must determine whether modulo address-updates will be required to make the buffer pointer "wrap-around" the end of the buffer. Such modulo address updates normally require overhead; the amount of overhead varies from processor to processor. For instance, recall example 1, which illustrates the Motorola DSP56000's hardware support for modulo address generation. Here a "modifier register" must be loaded with the buffer size before modulo updates can be performed on the corresponding address register, so there is a potential overhead of one instruction every time the buffer pointer is swapped into the register file. When there is no hardware support for modulo addressing, as with general purpose RISC microprocessors such as the MIPS R3000 [17], the modulo update must be performed in software every time the buffer is accessed. A sample MIPS R3000 assembly code sequence to perform this update is shown in figure 10. This reveals an overhead of several instructions for each buffer access.

For static buffering, we know exactly which accesses require a modulo update. We need not perform modulo address computations for any other access, and for the accesses that wrap around the buffer, we can simply load the start address of the buffer into the corresponding address register — no explicit modulo computation is required. For example, consider the example in figure 11(a) and suppose that a buffer of length 4 is used. Then clearly the read-pointer for B wraps around the buffer between the first and second accesses of the second invocation B_2. Thus code for B_2 could have the structure outlined in figure 11(b). The only overhead for modulo buffering in this case is a single load instruction — regardless of whether or not hardware support for modulo addressing is required.

```
        lw $10, 22($11)        # # Load the address of the end of the buffer.
        beq $10, $12, $40      # # Compare with the buffer pointer.
        addi $12, 1            # # Increment pointer if not equal.......
        j $41
$40:
        lw $12, 23($11)        # # Otherwise, reset the pointer to the start of the buffer.
$41:
```

Fig 10. Sample MIPS R3000 assembly code to perform a modulo address calculation. Note that both registers and labels are identified by leading '$' characters.

With dynamic buffering, no invocation accesses the buffer at the same offset every schedule period. To see this, suppose some invocation A_j accesses a buffer β at the same offset every period. Since the buffer pointer for A_j advances TNSE positions from one schedule period to the next, it follows that TNSE must be a positive integer multiple β's logical buffer size, and thus the buffer must be static. Thus, absolute addressing is never possible for a dynamic buffer — *dynamic buffers must be contiguous*, and if an actor A accesses a dynamic buffer, the current position in the buffer must be maintained as a state variable of A. We find register-indirect addressing most appropriate, and when available, hardware autoincrement/autodecrement should be used to advance the buffer pointer in parallel with the accesses.

Another important aspect of the physical layout of a buffer is the effect on total storage requirements. The locations of a scattered buffer can be considered as independent entities with respect to memory allocation, and graph coloring [12] can be used to assign physical memory locations to the set of scattered buffers. If all scattered buffers correspond to *delayless* arcs then the interference graph becomes an interval graph, and interval graphs can be colored with the minimum number of colors in linear time [31, 5]. The presence of delay on one more of the relevant arcs complicates coloring substantially. A delay results in a sample that is read in a schedule period after the period in which it is written, and thus the lifetime of the sample crosses one or more iterations of the program's outermost (infinite) loop. The resulting interference graphs belong to the class of circular-arc graphs [13]. Finding a minimum coloring for this class of graphs is intractable, but effective heuristics have been developed [13].

When subsets of variables must reside in contiguous locations, we expect that the memory requirements will increase since this imposes additional constraints on the storage allocation problem. Until further insight is gained about this effect or a large set of experimental data is obtained, we cannot accurately estimate how much more memory will be required if a particular scattered buffer is changed to a contiguous buffer. However, since optimal storage layout requires scattered buffers, it is likely that when data-memory requirements are severe, arcs should be implemented as scattered buffers whenever possible. We will discuss storage optimization further in section 5.

schedule which takes full advantage of the looping possibilities for this graph is A(4B). However, this schedule results in five samples on the arc after A is fired, which exceeds the TNSE of four. Grouping all four invocations of B together in the schedule requires that the maximum number of samples on the arc exceed the TNSE.

4.3 Contiguous vs. Scattered Buffering

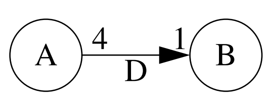
Once we have decided whether a buffer is to be static or dynamic, we may decide upon whether it will be a *contiguous buffer*, occupying a section of successive physical memory locations, or whether the buffer may be *scattered* through memory. The decision primarily affects the addressing modes that can be used to access the buffer and the storage efficiency of the memory allocation. Clearly, only a contiguous buffer can be accessed through register autoincrement/autodecrement indirect addressing, and thus a buffer that is accessed from within any kind of loop — a loop arranged by the scheduler or a loop that appears inside the code template for an actor — must usually be implemented using a contiguous buffer. The only exception occurs when all CCSS's associated with the source or sink of an arc access the arc statically — in this case absolute addressing can be used. Depending on the target processor, this may be an important exception to consider. For programmable DSPs such as the Motorola 56000, arbitrary absolute addresses require an additional word of program memory, and thus an additional instruction cycle. Register-indirect accesses require no such overhead and can often be performed in parallel with other operations [26]. Under these circumstances, contiguous buffering and register-deferred addressing are preferable for multiword buffers even if the loop-structure permits absolute addressing. On the other hand, many general-purpose RISC microprocessors allow large absolute displacements to be accessed through single-word instructions [14], but they do not allow register-indirect accesses to be issued in parallel with other instructions. Furthermore, they do not support autoincrement mode in hardware — a separate instruction must be issued to increment the index register. In this case there is no advantage to using register-indirect addressing when the loop structure does not require it. There is no point in incurring the overhead to initialize the address register and the overhead due to a possible increase in register swapping, and thus absolute addressing is preferable.

To fully asses the benefit of choosing static buffering over dynamic buffering for a particular arc, we must consult the first-reaches table at every spill-point. Performing this check on every multiword buffer is very expensive. Instead, we should perform this check only for critical sections of the program. For example if an arc carries a large amount of traffic, we would wish to choose dynamic buffering unless the arc is accessed from very frequently executed parts of the program and the loop structure permits taking advantage of static buffering. Similarly, the data-memory savings of implementing a low-traffic arc as a dynamic buffer is often negligible — the compiler has little to lose by choosing static buffering for such cases.

We conclude this subsection with a note on the requirements for static buffering. The two conditions of theorem 1 together imply that static buffering cannot be possible if TNSE is less than the maximum number of samples that coexist on the arc. For this to happen, clearly the arc must have nonzero delay since TNSE samples are produced and consumed from the arc every schedule period. When there is delay, however, it is possible that at some point in the schedule period, the arc will buffer more than TNSE tokens. For example, looping often creates a situation in which an arc must be implemented as a dynamic buffer. This is illustrated in figure 9. The

	A_1 A_2 A_3 A_4	C_1	B_1 B_3	C_2 C_3 C_5 C_6	B_2 B_4	C_4 C_7	B_5 B_6	C_8 C_9
A_1,A_2,A_3,A_4	T	T	T	F	F	F	F	F
C_1	T	F	T	T	F	F	F	F
B_1,B_3	T	F	F	T	T	F	F	F
C_2,C_3,C_5,C_6	T	F	F	T	T	T	F	F
B_2,B_4	T	F	T	F	F	T	T	F
C_4,C_7	T	F	T	T	F	F	T	T
B_5,B_6	T	F	T	F	F	F	T	T
C_8,C_9	T	T	T	F	F	F	T	T

Table 2. The *first-reaches* table associated with the looped schedule (4A)C(2B(2C)BC)(2BC) (the corresponding flow graph is shown in figure 5. The entry corresponding to a row CCSS X and a column CCSS Y is “true” (T) if and only if there is a control path that goes from X to Y without passing through another CCSS for the actor that corresponds to Y.



Schedule: A(4B)

Fig 9. An illustration of how looping can necessitate dynamic buffering.

associated with actor A and arc α must be swapped out of its register at some point in the program. First we must determine location X in the CCSS graph that corresponds to this swap-point. From X, we traverse all forward paths until they either reach the end of the program, they traverse the same node twice (they traverse a cycle), or they reach an occurrence of a CCSS for A. We are interested only in the *first* time a forward path encounters a CCSS for A. Let P be the set of all forward paths p from X which reach a CCSS for A before traversing any node twice, and let A(p) denote the first CCSS for A that p encounters. Then the buffer pointer must be spilled to memory if and only if the set P contains a member p^* such that $A(p^*)$ does not access α statically.

Traversing forward paths at every spill may be extremely inefficient. Instead, we can perform a one-time analysis of the loop organization to construct a table containing the desired reachability information. The concept is similar to the conventional global data flow analysis problem of determining which variable definitions reach which parts of the program [1]. However, our problem is slightly more complex. In global dataflow analysis, we need to know which variable definitions are live at a given point in the program. For eliminating buffer-pointer spills, we need to know which points in a program can reach a given CCSS *without passing through another CCSS for the same actor*. This information can be summarized in a boolean table which has each entry indexed by an ordered pair of CCSS's (C_1, C_2). The entry for (C_1, C_2) will be true if and only if there is a control path from C_1 to C_2 which does not pass through another CCSS for the actor that corresponds to C_2. We refer to this table as the *first-reaches* table since it indicates the points (the CCSS's) at which control first reaches a given actor from a given CCSS.Table 2 shows the first-reaches table for the looped schedule (4A)C(2B(2C)BC)(2BC). The CCSS flow graph associated with this schedule is depicted in figure 5.

In the appendix, we describe a technique for constructing the first-reaches table based largely on methods described in [1] for reaching definitions. An important difference is that a separate pass through the loop hierarchy is required to construct the columns associated with each actor, whereas reaching definitions can be dealt with in a single pass. In practice, however we are concerned only with the columns of the first-reaches matrix that correspond to actors which access multiword contiguous buffers, so often a large number of passes can be skipped.

schedule (4A)C(2B(2C)BC)(2BC) for this SDF graph. We can tabulate the offsets for every buffer access in the program to examine the access patterns for each CCSS. Such a tabulation is shown in table 1, assuming that static buffers of length 12 and 6 are used for arcs A↑B and B↑C respectively. The *access port* column specifies the different node-arc incidences in the SDF graph. For example A → A↑B refers to the connection of actor A to the *input* of arc A↑B (the side without the arrowhead), and B↑C → C refers to the connection of the output of arc B↑C (the side with the arrowhead) to actor C. The *invocation* column lists the firings of the actor with the associated access port, and the offset at which the *i*th invocation of this actor references the access port is given in the *i*th *offset* entry for the access port. Examination of table 1 reveals that the members of CCSS $\{C_4, C_7\}$ read from arc B↑C at the same offset. Similarly the write accesses of CCSS's $\{B_1, B_3\}$ and $\{B_2, B_4\}$ occur respectively at the same offsets. If all members of a CCSS X access an arc α at the same offset, we say that X **accesses** α **statically**.

Thus when a pointer into a static buffer is spilled, and the pointer is accessed elsewhere from within a loop, it is not always necessary to spill the pointer to memory. The procedure for determining whether a spill is necessary at a given swap point can be conceptualized easily in terms of the CCSS flow graph, which we introduced in section 2. Suppose that a buffer pointer

access port	invocation	offset
A↑B→ B	1	0
	2	2
	3	4
	4	6
	5	8
	6	10
B → B↑C	1	3
	2	0
	3	3
	4	0
	5	3
	6	0

access port	invocation	offset
B↑C→ C	1	0
	2	2
	3	4
	4	0
	5	2
	6	4
	7	0
	8	2
	9	4
A → A↑B	1	0
	2	3
	3	6
	4	9

Table 1. A tabulation of the buffer access patterns associated with the schedule (4A)C(2B(2C)BC)(2BC) for the SDF graph in figure 3(b).

memory so that the next time the pointer is used, it can resume from the correct position in the buffer. With static buffering, we know the offset at which every invocation accesses the buffer. Thus we can resume the buffer addressing with an immediate value and there is no need to spill the pointer to memory. The net result is that every time a buffer pointer of the source or sink node is swapped out, dynamic buffering requires an extra store to memory.

However, looping may limit the savings in overhead for static buffering. For instance, consider the example in figure 8. It can easily be verified that the repetitions counts for A, B, C, D, and E are respectively 1, 2, 4, 4, and 4 invocations per schedule period. Since TNSE($B\uparrow C$) = 4, a buffer of size four suffices for static buffering on the arc between B and C. Now the code block for C must access $B\uparrow C$ through some physical address register R, and R must contain the correct buffer position C_{rp} every time the code block is entered. If it is not possible to dedicate R to C_{rp} for the entire inner loop (2DCE), then R must be loaded with the current value of C_{rp} just prior to entering the code block for C. Since the code block executes C_1, C_2, C_3 and C_4 — the members of the associated CCSS — and each of these invocations accesses the buffer at a different offset, we cannot load R with an immediate value. R must be obtained from a memory location and the current value of C_{rp} must be written to this location whenever R is swapped out. It can easily be verified that at most three samples coexist on $B\uparrow C$ at any given time, and thus a dynamic buffer of size three could implement the arc. Since the organization of loops precludes exploiting the static information of a length four buffer, dynamic buffering is definitely preferable in this situation.

It is not always the case, however, that different members of a CCSS access a static buffer at different offsets. As an illustration of this, consider again the example in figure 3(b), and the

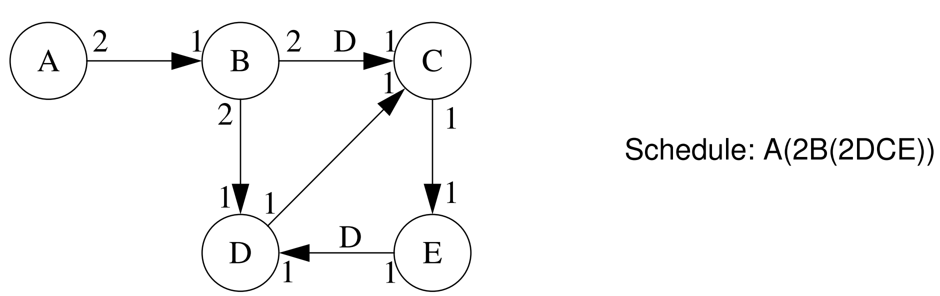


Fig 8. An example of how loops can limit the advantages of static buffering.

ing arc α

2. If α has no delay, then static buffering is possible with any buffer size that meets criterion 1. Otherwise, static buffering is possible if and only if TNSE is a positive-integer multiple of N.

Thus, static buffering for an arc with delay may require additional storage space — 50% more in the case of the example in figure 7. The difference may be negligible for most buffers, but it must be kept in mind when sample rates are very high. The storage economy of non-static, or *dynamic*, buffering comes at the expense of potential execution-time overhead. When a pointer to a dynamic buffer is swapped out of its physical register, it is mandatory that its value be spilled to

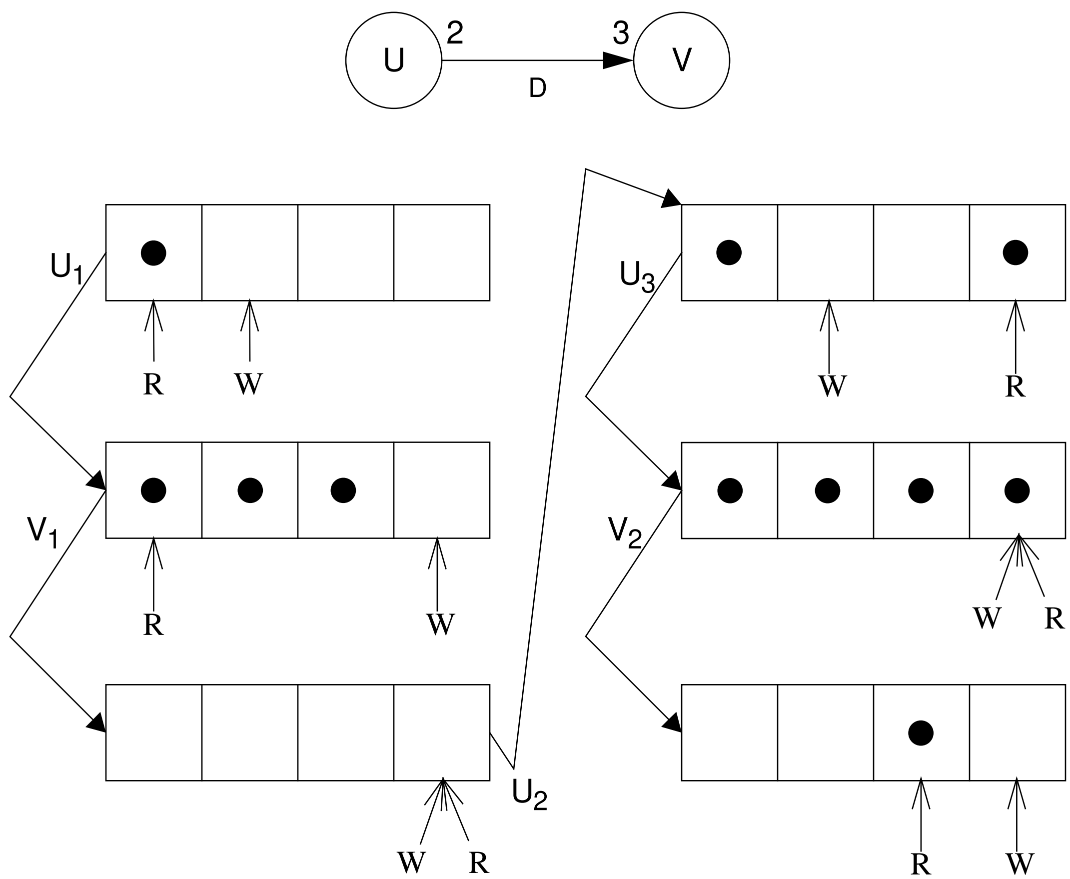


Fig 7. The effect of delay on the minimum buffer size required for static buffering. With a buffer size of only 4, the location of the “delay sample” shifts two positions each schedule period. The schedule in this example is UVUUV.

then we will occasionally denote α by "source(α)$\uparrow$sink(α)". For example U$\uparrow$V denotes the arc from U to V in figure 6.

4.2 Static vs. Dynamic Buffering

The first quality of a buffer that should be decided upon is whether or not the buffer is static. For an SDF arc α, static buffering means that for both source(α) and sink(α), the *i*th sample accessed in any schedule period resides in the same memory location as the *i*th sample accessed in any other schedule period [23]. From our discussion of figure 6, it is clear that when there is no delay on α, static buffering can occur with a logical buffer size equal to the maximum number of live samples that coexist on the arc. However, if α has nonzero delay, then we must impose an additional constraint that *TNSE is some positive integral multiple of the buffer length.* A "delay" on α can be viewed simply as an initial sample. In steady state, it can be viewed as data produced in one schedule period and consumed in the next.

The need for this constraint is illustrated in figure 7. Here, the minimum buffer size according to the previous rule is four, since up to four samples can concurrently exist on the arc. Figure 7 shows the succession of buffer states if a buffer of this length is used. Now since there is a delay on the arc, there will always be a sample in the buffer at the beginning of *each* schedule period — this is the first sample consumed by V_1. For static buffering, we need this *delay sample* — which is consumed in the schedule period *after* it is produced — to reside in the same memory location every period. Comparison of the initial and final buffer states in figure 7 reveals that this is not the case, since the write pointer W did not wrap around to point to its original location. Clearly, W could have returned to its original position if and only if the total number of advances made by W (6, in this case) was an integer multiple of the buffer length. But the total number of advances made by W is simply TNSE.

We have motivated the following theorem:

Theorem 1: For a given schedule, the logical buffer size N must satisfy the following conditions

1. N cannot be less than the maximum number of live samples which coexist on the correspond-

4 A CLASSIFICATION OF BUFFERS

We must determine four qualities of a buffer to guide memory allocation and code generation — the *logical size* of the buffer, whether the buffer will be contiguous, whether the accesses to the buffer are static, and whether the buffer is circular or linear. By the logical size of a buffer, we mean the number of memory locations required for the buffer if it is implemented as a single contiguous block of memory. For example, the buffer for the graph of figure 6 will have a logical size of four or six depending, respectively, on whether or not we are willing to pay the cost of resetting the buffer pointers before the beginning of every schedule period. In section 5, we will show that it may often be desirable to implement a buffer in multiple nonadjacent segments of physical memory. We will also show, however, that in such cases, the logical buffer size parameter is still important for guiding the memory allocation process.

4.1 Terminology

We digress briefly to introduce some definitions and notation that will be used frequently throughout the rest of this paper.

We use the following notation to express the parameters of an SDF arc α:

- source(α) = the source node of α.
- sink(α) = the sink node of α.
- p(α) = the number of samples produced onto α each time source(α) is invoked.
- c(α) = the number of samples consumed from α each time sink(α) is invoked.
- delay(α) = the delay on α.

We define the *total number of samples exchanged on* α — abbreviated TNSE(α) or just TNSE, when the arc in question is understood — to be the total number of samples produced onto α by source(α) during a schedule period, or equivalently the total number of samples consumed from α during a schedule period. Finally, if α is the only arc directed from source(α) to sink(α),

Observe also that the pointers R and W can be reset at the beginning of each schedule period to point to the beginning of the buffer, and thus the access patterns depicted in figure 6 could be repeated every period. This would cause the locations in each buffer's access to be *static* — fixed for every iteration of the periodic schedule — and hence they would be known values at compile time.

This illustration renders false the previous notion that for static buffering, the total number of samples exchanged on an arc per schedule period must always be a multiple of the buffer size. As we will show in the following section, the requirement holds only when there is a nonzero delay associated with the arc in question.

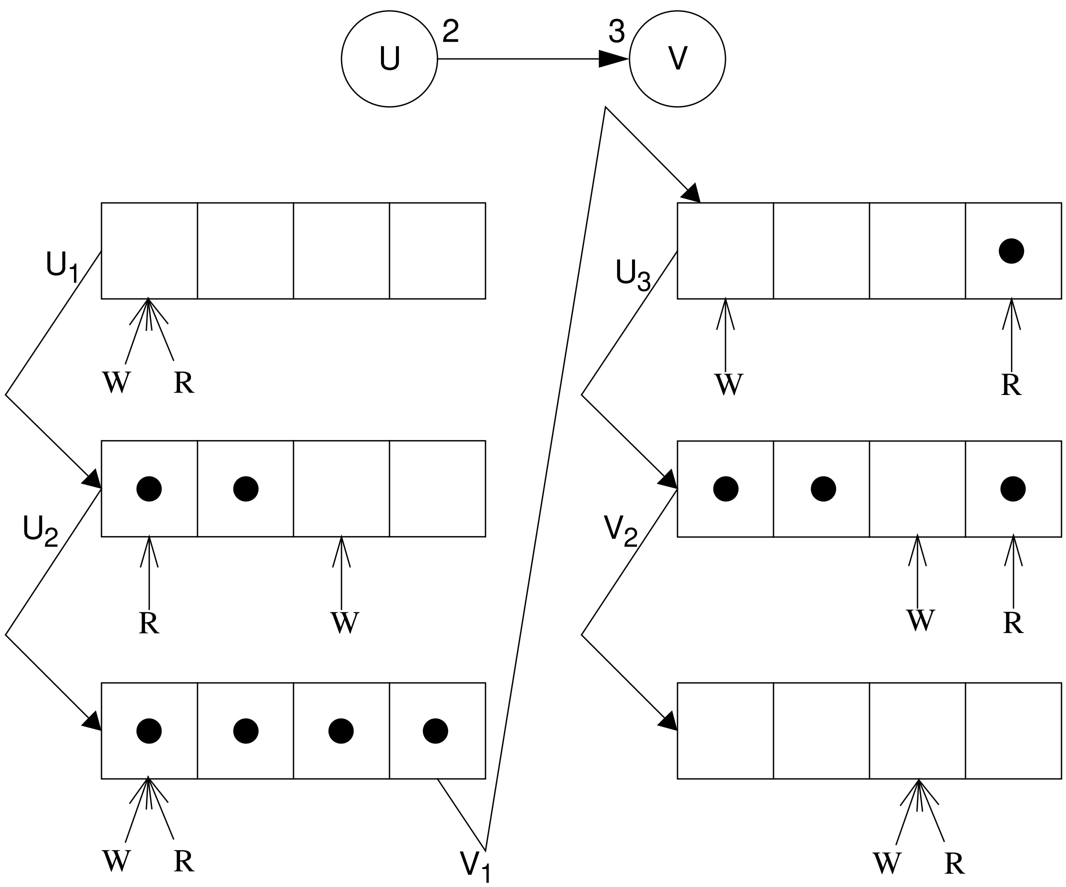


Fig 6. An illustration of modulo addressing. This figure shows how the position of samples in a buffer changes as the firings in a schedule are carried out. The schedule in this example is U(2UV). "W" and "R" represent the write pointer for U and the read pointer for V respectively.

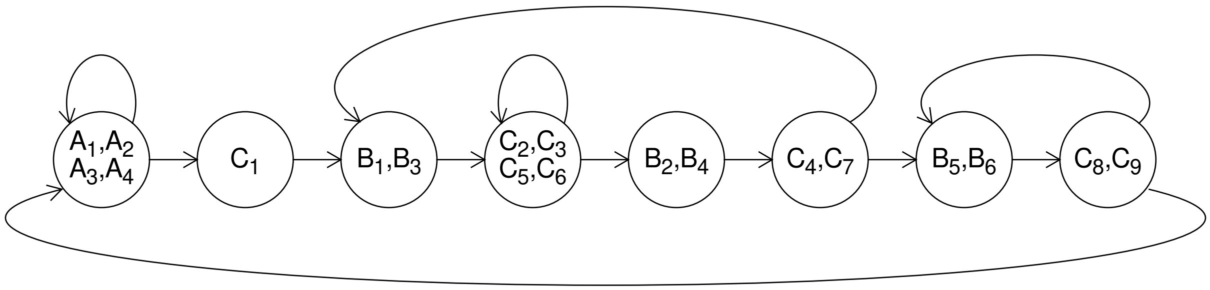


Fig 5. The CCSS flow graph associated with the schedule (4A)C(2B(2C)BC)(2BC) for the SDF graph in figure 3(b).

accesses to be sequential. This addressing mode allows for efficient implementation of circular buffers, for which indices need to be updated modulo the length of the buffer so that they can wrap around to the other end. For example, consider the modulo addressing support provided in the Motorola DSP56000.

Example 1: In the Motorola DSP56000 programmable DSP, a modifier register MX is associated with each address register RX. Loading MX with a value $n > 0$ specifies a circular buffer of length $n + 1$. The starting address of the buffer is determined by the value V that is stored in RX. If we let B denote the value obtained by clearing the $\lceil \log_2 [n+1] \rceil$ least significant bits of V, then assuming that $B \le V \le (B + n)$, an autoincrement access (RX)+ updates RX to $\{B + [(V - B + 1) \bmod (n + 1)]\}$.

Figure 6 illustrates the use of modulo addressing to decrease memory requirements when sequential buffer access is needed. The schedule U(2UV) would clearly require a buffer of size 6 for iterative access if only linear addressing is available. However, as the sequence of buffer diagrams in figure 6 shows, only four buffer locations are required when modulo addressing is used. W and R respectively denote the write pointer for U and the read pointer for V, and a black circle inside a buffer slot indicates a **live sample** — a sample which has been produced but not yet consumed. Note that the accesses of the second invocation of U and the second invocation of V wrap around the end of the buffer.

data using the register autoincrement or autodecrement indirect addressing modes, addressing modes that were designed precisely for this purpose of iteratively stepping through successive items of data. Here, the outputs of A are stored to successive locations *buf* and *buf+1*, and B reads these values into local register x0 through the autoincremented buffer pointer r2.

We conclude this section by introducing two definitions which will be useful throughout the remainder of the paper. The first definition provides a mapping from the appearances of actors in a looped schedule to the firings that they represent. In other words, it maps a code block in the target program to the set of invocations which it will execute.

Definition 1: Given an SDF graph G, a looped schedule S for G, and a node A in G, a **common code space set**, abbreviated **CCSS**, for A is the set of invocations of A which are represented by some appearance of A in S.

A CCSS is thus a set of invocations carried out by a given sequence of instructions in program memory (code space). For example consider the looped schedule (4A)C(2B(2C)BC)(2BC) for the SDF graph in figure 3(b). The CCSS's for this looped schedule are $\{A_1, A_2, A_3, A_4\}$, $\{C_1\}$, $\{B_1, B_3\}$, $\{C_2, C_3, C_5, C_6\}$, $\{B_2, B_4\}$, $\{C_4, C_7\}$, $\{B_5, B_6\}$, and $\{C_8, C_9\}$.

It will be useful to examine the *flow* of common code space sets. This can be depicted with a directed graph, called the **CCSS flow graph**, that is largely analogous to the *basic block* graph [1] used in conventional compiler techniques. Each CCSS corresponds to a node in the CCSS flow graph, and an arc is inserted from a CCSS A to a CCSS B if and only if there are invocations $A_i \in A$ and $B_j \in B$ such that B_j is fired immediately after A_i. To illustrate CCSS flow graph construction, figure 5 shows the CCSS flow graph associated with the schedule (4A)C(2B(2C)B-C)(2BC) for the SDF graph in figure 3(b).

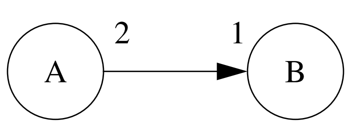
3 MODULO ADDRESSING

Most programmable DSP's offer a *modulo addressing mode*, which can be used in conjunction with careful buffer sizing to alleviate the memory cost associated with requiring buffer

firing sequence X_1 X_2 … X_M. These compact representations of the two schedules reveal that they are two distinct nested loop organizations for the same graph. It is important for a scheduler to recognize this distinction because the buffering requirements may vary significantly. In this case, for example, the former schedule requires 27 words of data memory and the latter schedule requires 21.

In [2], we discuss the problem of scheduling SDF graphs to effectively synthesize looping from iteration. When there is a large amount of iteration, these techniques may be crucial to reducing the code-space requirements to a level that will allow the program to fit on-chip. Thus we must examine the code-generation aspects of having loops in the target code.

The primary code generation issue for loops is the accessing of a buffer from within a loop. The difficulty lies in the requirement for different invocations of the same actor to be executed with the same block of instructions. As a simple example, consider figure 4, which shows a multirate SDF graph, a looped schedule for the graph, and an outline of Motorola DSP56000 assembly code that could efficiently implement this schedule. In the code outline, the statement "do #N LABEL" specifies N successive executions of the block of code between the "do" statement and the instruction at location LABEL. Thus the successive firings of B are carried out with a loop. This requires that both invocations of B must access their inputs with the same instruction, and that the output data for A be stored in a manner that can be accessed iteratively. This in turn suggests writing the data produced by A to successive memory locations, and having B read this



Schedule: A(2B)

```
code for "A"
outputs in x0 and y0
move x0, buf
move y0, buf + 1
move #buf, r2
do #2, LOOPEND
move (r2)+, x0
code for "B"
input in x0
LOOPEND:
```

Fig 4. An illustration of compiled code for a looped schedule.

times per schedule period and B must fire at least twice. It is thus natural to define iteration in multirate SDF as the change in firing-rate which is manifested by a change in the production and consumption rates along an arc[20].

In conventional programming languages, the notion of iteration is normally associated with loops, in which the programmer specifies that a sequence of code is to be repeated some number of times *in succession*. However, in SDF there are three mechanisms which force us to distinguish looping from iteration. The most fundamental reason is that an SDF graph specifies only a partial ordering on the computations involved. Whether or not repeated firings are invoked in succession depends on how the graph is scheduled. Second, feedback constraints may restrict the degree of looping that can be assembled from an instance of iteration. For example, figure 3(a) shows a multirate SDF graph that consists of a simple feedback loop. The only possible periodic schedule for this graph is BAB, which offers no opportunity for looping within a single schedule period. If, however, the delay on the lower arc were transferred to the upper arc, or if the upper arc were removed, then the sample-rate change between A and B could be translated into the schedule BBA, which allows a loop to subsume the firings of B. Finally, a cascade of iterations, the SDF form of *nested iteration* [20], does not translate into a unique opportunity for nested loops. For example, two possible schedules for the graph in figure 3(b) are AABBBAABBBCCCCCCCCC. and AAAABBCCCBBCCCBBCCC. Using the **looped schedule** notation defined in [2], we can express these schedules more compactly as (2 (2A) (3B)) (9C) and (4A) (3 (2B) (3C)) respectively. Here each parenthesized term (N X_1 X_2 … X_M) represents N successive invocations of the

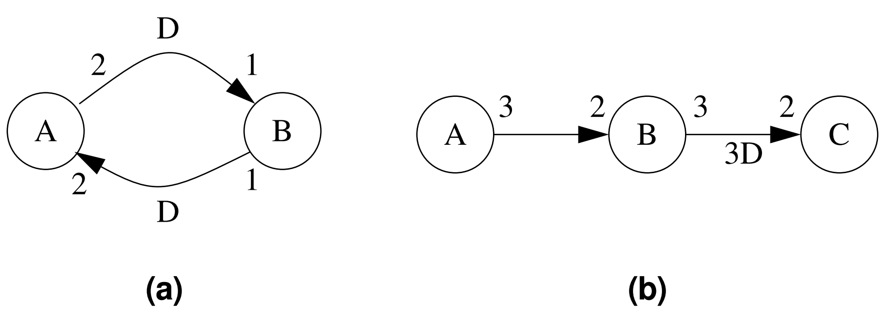


Fig 3. Examples that illustrate distinctions between iteration and looping in SDF.

Systematically incorporating buffering considerations in the scheduling process is a topic that we are currently examining.

We begin by reviewing the scheduling and code generation issues involved in effectively organizing loops in the target code. In section 3 we discuss circular buffers, which play a key role in multirate buffering. Section 4 presents a classification of buffers based on dataflow properties and discusses tradeoffs between the different categories. Most buffer-related optimizations apply only to particular subsets of these categories. The following section examines the problem of overlaying buffers for compact memory allocation. Section 6 considers optimization opportunities that apply to modulo buffers. Section 7 describes a class of actors that can be implemented very efficiently by abandoning their dataflow interpretation and using more intelligent buffering. Finally, section 8 presents concluding remarks.

Although the techniques in this paper are presented in the context of block-diagram programming, they can be applied to other DSP design environments. Many of the programming languages used for DSP, such as Lucid[30], SISAL[24] and Silage[10] are based on or closely related to dataflow semantics. In these languages, the compiler can easily extract a view of the program as hierarchy of dataflow graphs. A coarse level view of part of this hierarchy may reveal SDF behavior, while the local behavior of the macro-blocks involved are not SDF. Knowledge of the high-level synchrony can be used to apply “global” optimizations such as those described in this paper, and the local subgraphs can be examined for finer SDF components. For example, in [7], Dennis shows how recursive stream functions in SISAL-2 can be converted into SDF graphs. In signal processing, usually a significant fraction of the overall computation can be represented with SDF semantics, so it is important to recognize and exploit SDF behavior as much as possible.

2 Multirate Code Generation Issues

If the number of samples produced on an SDF arc (per invocation of the source actor) does not equal the number of samples consumed (per sink invocation), the source actor or the sink actor must be repeated, and when the number of samples produced and consumed form a nonintegral ratio, both actors must be repeated. For example, in figure 2, actor A must fire at least three

looping. For example, for the graph in figure 2, (AABCABC) and (AAABCBC) are both permissible periodic schedules. The latter schedule clearly offers simpler looping, however the amount of memory required to implement the arc between A and B is 50% greater (600 words vs. 400 words). In general, increasing the degree of looping in the schedule significantly increases buffering requirements [2]. Thus, due to the limited amount of on-chip data memory in programmable DSPs, it is highly desirable to overlay noninterfering buffers in the same physical memory space as much as possible. This paper presents ways to analyze the dataflow information to detect opportunities for overlaying buffers which can be incorporated into a best-fit memory allocation scheme.

Normally, when an SDF graph G is compiled, the target program is an infinite loop whose body executes one period of a periodic schedule for G. We refer to each period of this schedule as a *schedule period* of the target program. In [22], it is shown that for each node N in G, we can determine a positive integer $q(\mathrm{N})$ such that every valid periodic schedule for G must invoke N a multiple of $q(\mathrm{N})$ times. More specifically, associated with each valid periodic schedule S for G, there is a positive integer J, called the *blocking factor* of S, such that S invokes every node M exactly $\mathrm{J}q(\mathrm{M})$ times. Thus, code generation begins by determining $q()$, selecting a blocking factor and constructing an appropriate schedule.

Several scheduling problems for SDF and related models have been addressed: constructing efficient multiprocessor schedules is discussed in [27, 29]; Ritz et. al discuss vectorization [28]; the problem or organizing loops is examined in [2]; and compiler scheduling techniques for efficient register allocation are presented in [26]. In this paper, we assume that a schedule has been constructed under one or more of these criteria. In other words, the techniques of this paper do not interact with the scheduling process — we assume that the schedule is fixed beforehand.

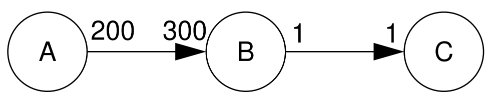


Fig 2. A multirate SDF graph for which looping greatly increases buffering requirements.

is assembly code for the Motorola DSP56000. The numbers adjacent to the inputs and the output represent the number of tokens consumed or produced each time the actor is invoked. In this example, “input1” and “input2” represent memory addresses where the operands to the addition actor are stored, and “output” represents the location in which the output sample will be buffered.

In figure 1, observe that four instructions are required to implement the addition actor. Simply augmenting the compiler with a register allocator and a mechanism for considering buffer locations as candidates for register-residence can reduce the cost of the addition to three, two or one instruction. The Comdisco Procoder graphical DSP compiler [26] demonstrates that integrating buffering with register allocation can produce code comparable to the best manually-written code.

The Comdisco Procoder’s performance is impressive, however the Procoder framework has one major limitation: it is primarily designed for *homogeneous* SDF, in which a firing must consume exactly one token from each input arc and produce exactly one token on every output arc. In particular, it becomes less efficient when multiple sample rates are specified. Furthermore, their techniques apply only when all buffers can be mapped *statically* to memory. In general, this need not be the case, and we will elaborate on this topic in section 2.

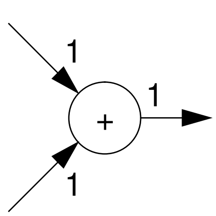
In this paper, we develop compiler techniques to optimize the buffering of multiple sample-rate SDF graphs. Multirate buffers are often best implemented as contiguous segments of memory to be accessed by indirect addressing, and thus they cannot be mapped to machine registers. Efficiently implementing such buffers requires reducing the amount of indexing overhead. We show that for SDF, there is a large amount of information available at compile-time which can be used to optimize the indexing of multirate buffers. Also, buffering and code generation for multirate graphs is complicated by the desire to organize loops in the target code. With large sample rate changes, failure to adequately exploit iteration may result in enormous code space requirements or excessive subroutine overhead. In [2], we develop techniques to schedule SDF graphs to maximize looping. We assume that such techniques are applied and examine the issues involved when buffers are accessed from within loops. Finally, multirate graphs may lead to very large buffering requirements if large sample rates are involved. This problem is compounded by

ing node in the dataflow graph consumes some number of data values (*tokens*) from each input arc and produces some number of tokens on each output arc. Dataflow imposes only partial ordering constraints, thus exposing parallelism. In Synchronous Dataflow (SDF), the number of tokens consumed from each input arc and produced onto each output arc is a fixed value that is known at compile time [23].

Another significant benefit of SDF is the ease with which a large class of signal processing algorithms can be expressed [3], and the effectiveness with which SDF graphs can be compiled into efficient microcode for programmable digital signal processors. This is in contrast to conventional procedural programming languages, which are not well-suited to specifying signal processing systems [10]. However, there are ongoing efforts towards augmenting such languages to make them more suitable; for example, [18] proposes extensions to the *C* language.

There have been several efforts toward developing compiler techniques for SDF and related models[11, 21, 26, 27, 28]. Ho [16] developed the first compiler for pure SDF semantics. The compiler, part of the Gabriel design environment [21], was targeted to the Motorola DSP56000 and the code that it produced was markably more efficient than that of existing *C* compilers. However, due to its inefficient implementation of buffering, the compiler could not match the quality of good handwritten code, and the disparity rapidly worsened as the granularity of the graph decreased.

The mandatory placement of all buffers in memory is a major cause of the high buffering overhead in Gabriel. Although this is a natural way to compile SDF graphs, it can create an enormous amount of overhead when actors of small granularity are present. This is illustrated in figure 1. Here, a graphical representation for an atomic addition actor is placed alongside typical assembly code that would be generated if straightforward buffering tactics are used. The target language




```
move input1, a
move input2, x0
add x0, a
move a, output
```

Fig 1. An illustration of inefficient buffering for an SDF graph.