IEEE Transactions on Signal Processing, Vol. 42, No. 5, May, 1994.

MEMORY MANAGEMENT FOR DATAFLOW PROGRAMMING OF MULTIRATE SIGNAL PROCESSING ALGORITHMS[1]

Department of Electrical Engineering and Computer Science

University of California

Berkeley, California 94720

Shuvra S. Bhattacharyya
Edward A. Lee

ABSTRACT

Managing the buffering of data along arcs is a critical part of compiling a synchronous dataflow (SDF) program. This paper shows how dataflow properties can be analyzed at compile-time to make buffering more efficient. Since the target code corresponding to each node of an SDF graph is normally obtained from a hand-optimized library of predefined blocks, the efficiency of data transfer between blocks is often the limiting factor in how closely an SDF compiler can approximate meticulous manual coding. Furthermore, in the presence of large sample-rate changes, straightforward buffering techniques can quickly exhaust limited on-chip data memory, necessitating the use of slower external memory. The techniques presented in this paper address both of these problems in a unified manner.

Key words: Dataflow Programming, Code Generation, Memory Allocation, Graphical Programming, Optimizing Compilers, Multirate Signal Processing.

1 Introduction

Dataflow [7] can be viewed as a graph-oriented programming paradigm in which the nodes, or *actors*, of the graph represent computations, and directed edges between nodes represent the passage of data between computations. A computation is deemed ready for execution whenever it has sufficient data on each of its input arcs. When a computation is executed, or *fired*, the corresponding node in the dataflow graph consumes some number of data values (*tokens*) from

[1] This research was sponsored by Defense Advanced Research Projects Agency (monitored by the U. S. Department of Justice, Federal Bureau of Investigation, under contract no. J-FBI-90-073), and by the National Science Foundation (MIP-9201605).

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].

Figure 1 shows examples of SDF graphs. Each arc is annotated with the number of samples produced by its source and the number of samples consumed by its sink. Thus, in (a), actor A produces two samples on its output arc each time it is invoked and B consumes one sample from its input arc. The "D" on each arc in (a) designates a unit delay, which we implement as an initial token on the arc. In the SDF-based design environments to which this paper applies, actors typically range in complexity from basic operations such addition or subtraction to signal processing subsystems such as FFT units and adaptive filters.

A significant benefit of SDF is the ease with which a large class of signal processing algorithms can be expressed [4], 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 [11]. 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[12, 21, 25, 26, 27]. 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

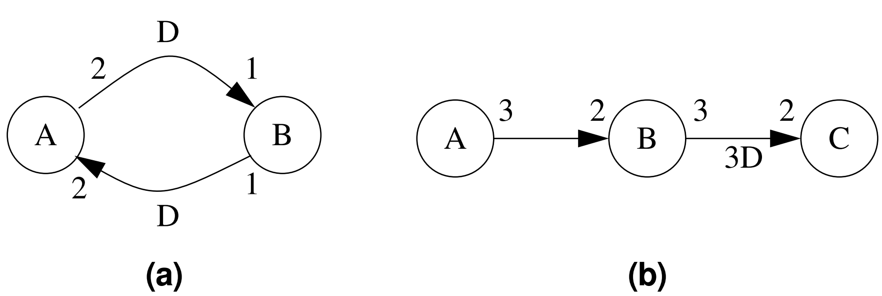


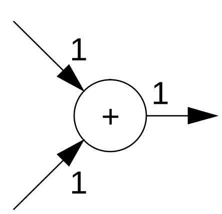
Fig 1. Examples of SDF graphs.

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 2. 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 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 [25] 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, the 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 4.




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

Fig 2. An illustration of inefficient buffering for an SDF graph.

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, multirate graphs may lead to very large buffering requirements if large sample rates are involved, and this problem is compounded by looping [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 "best-fit" and related memory allocation schemes.

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 $Jq(\mathrm{M})$ times. Thus, code generation begins by determining $q()$, selecting a blocking factor and constructing an appropriate schedule. The blocking factor can be viewed as the number of times that a minimal periodic schedule is executed through one iteration of the outermost loop that comprises the program. The blocking factor can affect the degree to which code can be vectorized, as discussed in [27]. Some other consequences of the choice of blocking factor in uniprocessor code generation are discussed later in this paper.

Several scheduling problems for SDF and related models have been addressed: constructing efficient multiprocessor schedules is discussed in [26, 28]; Ritz et. al discuss vectorization [27]; the problem or organizing loops is examined in [2]; and compiler scheduling techniques for efficient register allocation are presented in [25]. 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. Systematically incorporating buffering considerations into 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 modulo buffers, which play a key role in multirate buffering. Section 4 presents a classification of buffers based on dataflow properties and discusses these different categories with regards to storage requirements. The following three sections present code optimization techniques. Section 5 discusses minimizing spills of address register to memory. Section 6 examines the problem of overlaying buffers for compact memory allocation. Section 7 considers optimization opportunities that apply to modulo buffers. Finally, section 8 presents a detailed summary of the proposed methods.

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[29], SISAL[24] and Silage[11] are based on or closely related to dataflow semantics. In these languages, the compiler can easily extract a view of the program as a 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 [8], 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. Thus we 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 1(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 1(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 ($N\ X_1\ X_2 \ldots X_M$) represents N successive invocations of the firing sequence $X_1\ X_2 \ldots X_M$. 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 3, 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 data using the register autoincrement or autodecrement indirect addressing modes. 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.

The techniques in this paper do not depend on a specific language for defining the actors. However the techniques are best-suited when actor inputs and outputs are referenced symbolically, and the assignment of machine registers and memory locations is performed by the compiler, as in the Comdisco Procoder [25]. In this type of actor definition language, a simple adder actor might have the following as its code block:

```
add in1, in2, out
```

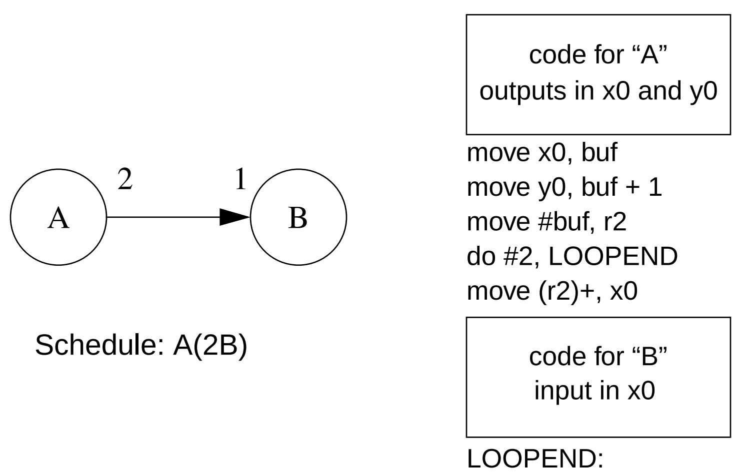


Fig 3. An illustration of compiled code for a looped schedule.

It is left to the compiler to replace “in1”, “in2” and “out” with register references and to make sure that data is routed appropriately between the registers. For example, if the adder is executed through a loop, and this loop does not contain the actor whose output is consumed by input port “in1”, it is generally desirable to load the register corresponding to “in1” through an address register. This is the case the with the input to actor B in figure 3. Alternatively, the schedule may permit data to be exchanged directly through registers, in which case the generated code might look like:

```
add r0, r1, r2
add r2, r3, r4
```

(this corresponds to a cascade of adders).

Another important code generation issue is register allocation, which is critical both for data and address registers. Scheduling heuristics for improving register allocation in homogeneous SDF block diagrams are discussed in [25]. These techniques can be applied to homogeneous subsystems in multirate graphs in conjunction with clustering techniques, such as those described in [2]. A recently-developed approach to register allocation studied by Hendren et. al [14] appears promising for multirate code generation. In this technique, a hierarchy of circular-arc graphs is extracted from nested loop code, and heuristics for coloring this class of graphs are applied. The techniques developed in this paper do not depend on a specific method of register allocation.

We conclude this section by introducing two definitions. 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 1(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 4 shows the CCSS flow graph associated with the schedule (4A)C(2B(2C)BC)(2BC) for the SDF graph in figure 1(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 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.

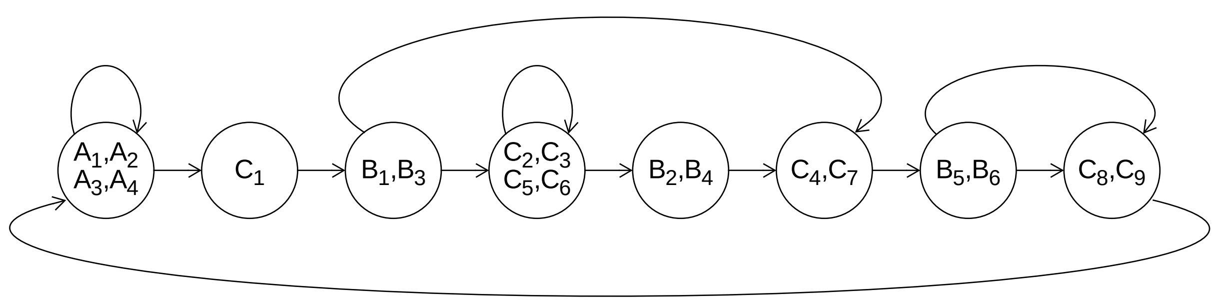


Fig 4. The CCSS flow graph associated with the schedule (4A)C(2B(2C)BC)(2BC) for the SDF graph in figure 1(b).

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 5 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 *postincrement* 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.

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 5

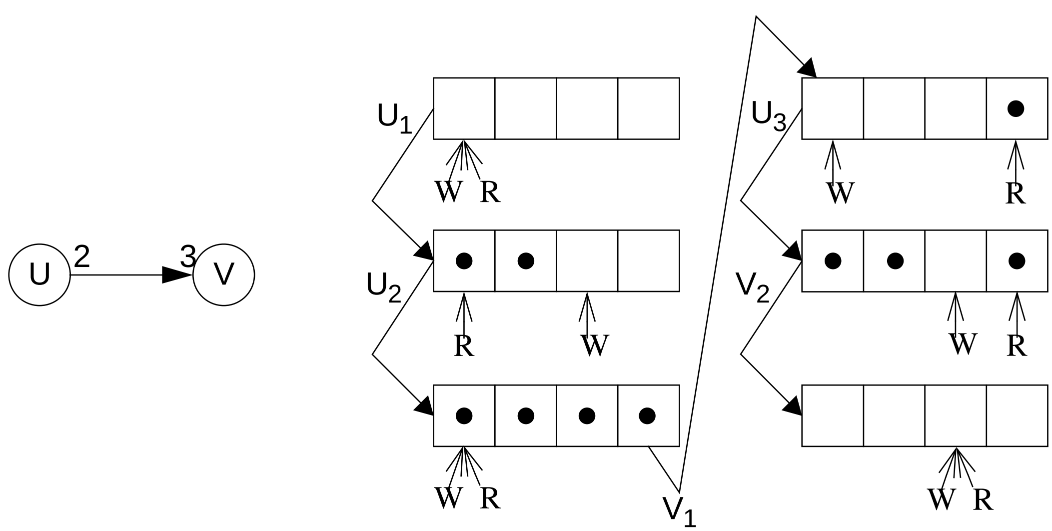


Fig 5. An illustration of modulo addressing. This figure shows how the position of samples in a buffer changes as the firings in a schedule are carried out. The schedule in this example is U(2UV). “W” and “R” represent the write pointer for U and the read pointer for V respectively.

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.

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 5 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 6, we will show that it may often be desirable to implement a buffer in multiple nonadjacent segments of physical memory.

Note that in our model of buffering, as in figure 5, each sample is read (consumed) from the same memory location that it is produced into, and thus there is no rearrangement of live data in the physical memory space.

4.1 Terminology

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), 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 5.

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]. A buffer that is not static is called a *dynamic* buffer. From our discussion of figure 5, 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.*

The need for this constraint is illustrated in figure 6. Here, the minimum buffer size according to the previous rule is four, since up to four samples can concurrently exist on the arc. Figure 6 shows the succession of buffer states if a buffer of this length is used. 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 6 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 summarize with 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 corresponding arc α.

2. If α has no delay, then static buffering is possible with any logical buffer size that meets criterion 1. Otherwise, static buffering is possible iff 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 6. The difference may be negligible for most buffers, but it must be kept in mind when sample rates are very high. Further tradeoffs between static and dynamic buffering are discussed in section 5.

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. Scattered buffering allows more flexibility in memory allocation, which can lead to lower memory requirements. However, as we discussed in section 2, contiguity constraints between the location of successive buffer accesses may be imposed by loops in the schedule. Similarly, loops that are contained in actor code blocks lead to contiguity constraints.

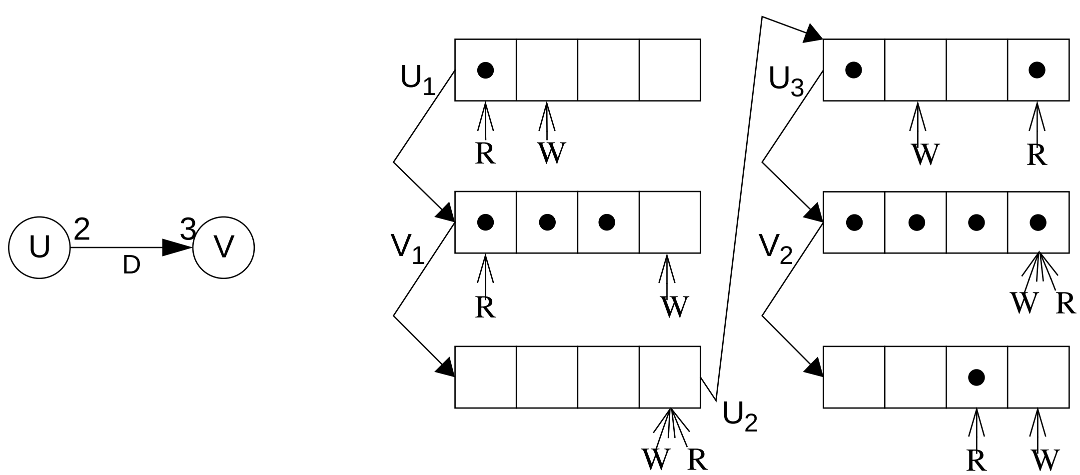


Fig 6. The effect of delay on the minimum buffer size required for static buffering. With a buffer size of only 4, the location of the “delay sample” shifts two positions each schedule period. The schedule in this example is UVUUV.

Dynamic buffering also induces contiguity constrains. In dynamic buffering, no invocation accesses the (logical) 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 of β's logical buffer size, and thus the buffer must be static. Thus, a dynamic buffer cannot be implemented with only absolute addressing — *dynamic buffers must be implemented with contiguous memory* (at least partially — for a complete treatment of the contiguity requirements for dynamic buffering, see [3]), 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.

An important aspect of the physical layout of a buffer is the effect on total storage requirements. The locations of a scattered buffer are not restricted to be mapped to continuous memory addresses, and graph coloring [13] 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 [6]. 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 [14]. Finding a minimum coloring for this class of graphs is intractable, but effective heuristics have been developed [14].

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 6.

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. This typically requires an overhead of several instructions for each buffer access.

In section 7, 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:

$$\mu(\alpha) = \left[\frac{TNSE(\alpha)}{\mathrm{minimum buffer size of}\,\alpha}\right] \times \left[\frac{1}{TNSE(\alpha) - (\mathrm{minimum buffer size of}\,\alpha)}\right]$$

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.

5 Increasing the Efficiency of Static Buffers

The storage economy of 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 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. As a result, every time a buffer pointer of the source or sink node is swapped out, dynamic buffering requires an extra store to memory.

For instance, consider the example in figure 7. 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 possi-

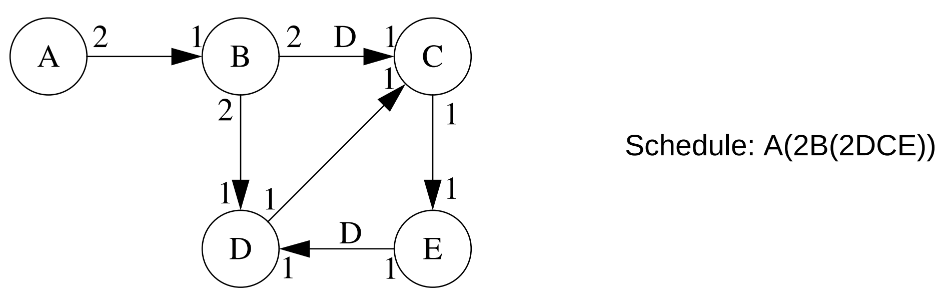


Fig 7. An example of how loops can limit the advantages of static buffering.

ble 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↑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 that different members of a CCSS access a static buffer at different offsets. As an illustration of this, consider again the example in figure 1(b), and the 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 arrow-

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 1(b).

head), and $B\uparrow C \rightarrow C$ refers to the connection of the output of arc $B\uparrow C$ (the side with the arrow-head) 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 $B\uparrow C$ 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 associated with actor A and arc α must be swapped out of its register at some point in the program. First we must determine the 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 4.

In [3], we specify 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.

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 sections of the program that are executed most frequently.

	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 CCSS flow graph is shown in figure 4). 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.

6 Overlaying Buffers

When large sample rate changes are involved, assigning each buffer to a single contiguous block of physical memory may require more data-memory space than what is available. In this section, we show how to fragment buffers in physical memory, which can expose more opportunities for overlaying [9]. This technique 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 [9] has studied more elaborate storage optimization schemes that incorporate a generalized interference graph. Such schemes are equally compatible with the methods developed in this section.

6.1 Fragmenting Buffer Lifetimes

Figure 8 illustrates how lifetime analysis and fragmentation information can be used to reduce storage requirements. Here, a multirate graph is depicted along with a looped schedule for

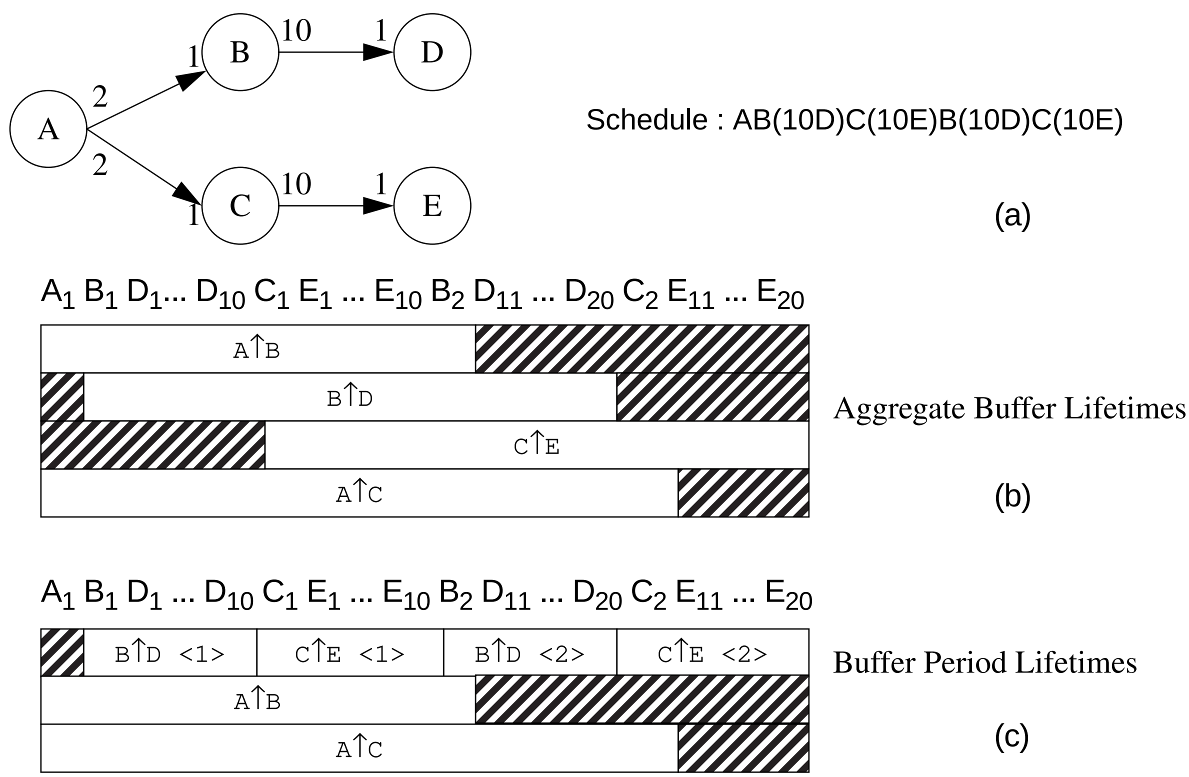


Fig 8. An illustration of opportunities to overlay buffers based on the periodicity of accesses.

the graph and the resulting buffer lifetime profiles. The first profile treats each arc as an indivisible unit with respect to storage allocation. We see that this straightforward designation of buffer lifetimes does not reveal any opportunity to share storage and thus A↑B, A↑C, B↑D and C↑E require 2, 2, 10 and 10 units of storage respectively, for a total of 24 units.

Notice, however, that invocations that access B↑D can be divided into two sets {B_1, D_1, D_2, …, D_{10}} and {B_2, D_{11}, D_{12}, …, 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 separate 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↑D, and we denote them by successive indices as B↑D<1> and B↑D<2>. The live range for C↑E can be decomposed similarly and the resulting lifetime profile is depicted in figure 8(c) (we 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 aggregate lifetimes of these arcs conflict, the buffer periods do not. Thus the memory requirements can be reduced almost in half to 14 words.

This fragmentation technique can be exploited by first-fit, best-fit, and related storage allocation schemes. In such schemes, we maintain a list of variables along with their sizes and lifetimes; if variable x becomes live earlier than variable y, then x occurs earlier in the list than y. Also, we maintain a free-list of unallocated contiguous segments of memory. At each step, we remove the head of the variable list from the list, and we assign it to a free memory block for the duration of the variable's lifetime. In first-fit allocation, we choose the first free block of sufficient size, while in best-fit, we choose the free block of sufficient size whose size differs from the size of the variable by the least amount. In general, best-fit leads to more compact allocation, while first-fit is computationally more efficient.

For example, if we use the aggregate buffer lifetimes in figure 8(b), then neither first-fit, best-fit, nor any other storage allocation scheme will achieve any overlaying between the four variables to be allocated, and 24 units of storage are required. On the other hand, the fragmented buffer information in (c) separates the items to be allocated into six variables. It can easily be verified that both first-fit and best-fit allocation require only 14 words of storage to achieve a valid storage layout for figure 8(c).

6.2 Computing Buffer Periods

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 a loop inside 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 9(a), actor A is programmed so that it writes its samples iteratively. The resulting contiguity constraints are illustrated in figure 9(b) — the three samples produced by each invocation must be stored in three adjacent memory locations. We specify these two constraints 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 9(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 9(c) to {A[6], A[1]}, {A[2], A[3]}, {A[4], A[5]}.

Determining the constraints due to schedule loops is also straightforward. Given an arc $A\uparrow B$ 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 $A\uparrow B$ which occur within L. We can derive these from the con-

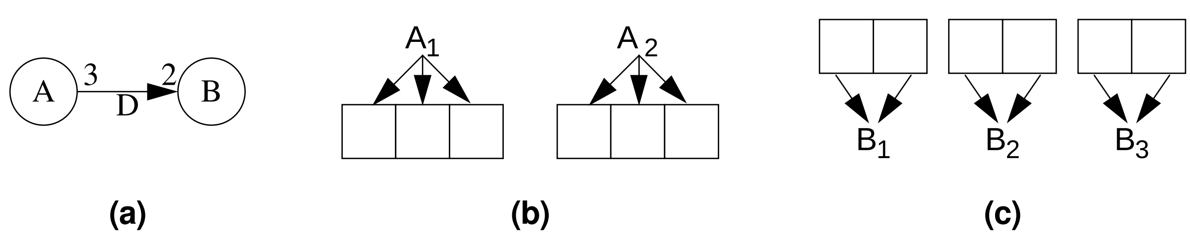


Fig 9. An illustration of buffering constraints when arcs are accessed through loops inside the actors.

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

tiguous 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.

So far we have only mentioned that dynamic buffering can also lead to constraint sets, but we have not fully described this effect. Due to limited space, we cannot derive the constraint sets for dynamic buffering here; instead we refer the reader to [3].

The constraint sets due to intra-actor looping, inter-actor looping and dynamic buffering together define the logical sections of a buffer that are restricted to contiguous segments of physical memory. 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.

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. $\left(\bigcup_{z=1}^{n} b_z\right) \cap \left[\bigcup_{E \in (C-b)} E\right] = \varnothing$.

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.

7 Eliminating Modulo Address Computations

In this section we develop a systematic approach to eliminating modulo accesses.

7.1 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

$$\rho_0 + n = (\text{some positive integer}) \times \mathrm{BUFSIZE},$$

where α denotes the arc in question; BUFSIZE denotes the length of the circular buffer; and ρ_0 denotes the buffer position of the initial access — i.e. $\rho_0 = \mathrm{delay}(\alpha)$ if we are concerned with the accesses of source(α) and $\rho_0 = 0$ if we are concerned with sink(α).

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. Proofs of these facts are given in [3].

Fact 1: Suppose a, b and c are positive integers, and suppose that a divides b and c. Then for some nonnegative integer k, $(b \bmod c) = ka$.

Fact 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 $(r + pk_2) \bmod q = k_1$.

Applying fact 1, with $a = \mathrm{gcd}(\mathrm{TNSE}, \mathrm{BUFSIZE})$, $b = \mathrm{k}_1\mathrm{TNSE}$, and $c = \mathrm{BUFSIZE}$, we see that for each positive integer k_1, there is a nonnegative integer k_2 such that

$$(k_1\mathrm{TNSE} \bmod \mathrm{BUFSIZE}) = k_2\mathrm{gcd}(\mathrm{TNSE}, \mathrm{BUFSIZE}).$$

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 ith access at offset j of window w_x, then, since the ith access shifts TNSE positions from schedule period to schedule period, we know that the ith access in any schedule period will occur at offset j of some window. For example, for the dynamic buffer in figure 10, it is easy to verify that for all schedule periods, the window offset for A's first access is 0.

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 fact 2 with $p = \mathrm{TNSE}/w_s$ = TNSE/gcd(TNSE, BUFSIZE); $q = n_w$ = BUFSIZE/gcd(TNSE, 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 2: Suppose α is an SDF arc, suppose N $\in$ {source(α), sink(α)}, and define ρ_0 = delay(α) if N = source(α), and $\rho_0 = 0$ if N = sink(α). Then the jth access ($j \in \{1, 2, \ldots, \mathrm{TNSE}\}$) of α by N is a wrap-around access in some schedule period iff

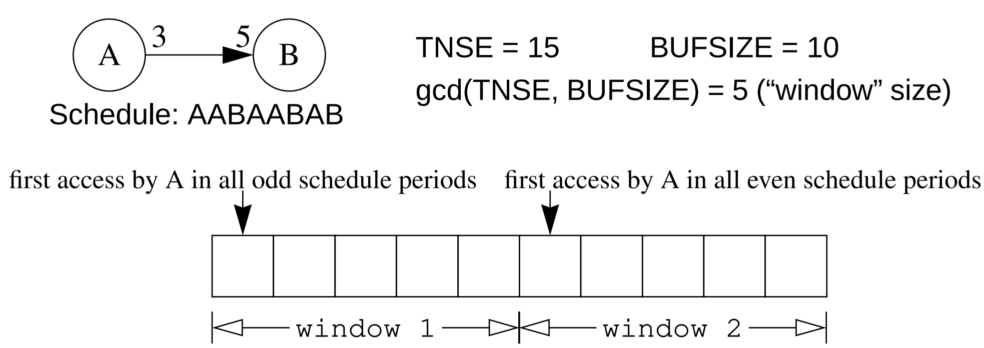


Fig 10. An illustration of repetitive access patterns in gcd(TNSE, BUFSIZE) windows.

$$[\rho_0 + (j - 1)] \bmod \gcd(\text{TNSE, BUFSIZE}) = \gcd(\text{TNSE, BUFSIZE}) - 1.$$

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:

$$j_W = \gcd(\text{TNSE, BUFSIZE}) - [\rho_0 \bmod \gcd(\text{TNSE, BUFSIZE})].$$

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

$$S_W = S_W(\alpha, \text{BUFSIZE}) = \left\{ j_W + n \times w_S \,\middle|\, n \in \{0, 1, \ldots, \text{floor}[(\text{TNSE} - 1) / w_S]\} \right\},$$

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

For the example of figure 10, 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 $n_W = \text{BUFSIZE} / \gcd(\text{TNSE, BUFSIZE})$ times.

We can reduce the average rate at which modulo computations must be performed by a factor of n_W if we increase the blocking factor to n_W. Assuming that all invocations of the same actor require the same amount of time to execute[1], the rate at which modulo computations must be performed is proportional to $R_M \equiv |S_w| / J$, where $|S_W|$ denotes the number of members in the set S_W, and J denotes the blocking factor. The denominator term J is required because the amount execution time required for a schedule period (an iteration of the target program's outermost loop) is proportional to the blocking factor. For example, in figure 10, $J = 1$, $S_W = \{5, 10, 15\}$, $|S_W| = 3$, and $R_M = 3$. If we increase the blocking factor to 2 and retain the same buffer size, $S_W = \{10, 20, 30\}$, $|S_W| = 3$, and $R_M = 1.5$ — thus the frequency of modulo address computations decreases by a factor of 2.

Observe that 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 10, a 7-word buffer can support the given schedule. However, this

1. In general this assumption does not hold; in such cases our analysis is not exact, but it gives a useful estimate.

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 frequently executed sections of code, it may be beneficial to explore tolerable increases in buffer size for the possible reduction of modulo updates.

7.2 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. Here we assume that the loop structure is fixed: 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 a given buffer access instruction sequence, the corresponding machine instruction(s) must perform a modulo address computation iff the associated set of accesses 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 2 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

$$[\rho_0 + (j-1)] \bmod \gcd(\mathrm{TNSE}, \mathrm{BUFSIZE}) = \gcd(\mathrm{TNSE}, \mathrm{BUFSIZE}) - 1.$$

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

8 Conclusions

Although the representation of multirate signal processing 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 has approached these problems in a unified manner and has developed systematic solutions to some of the significant problems. In this section we summarize the techniques developed, discuss related machine-dependent issues, and outline areas for further investigation.

We have presented a classification of buffers based on whether they are static or dynamic, linear or modulo, and contiguous or scattered; we evaluated the impact of these choices on storage requirements; and we have suggested guidelines for choosing between them. More thorough and systematic techniques to determine an optimal combination of buffering parameters is an important and challenging area for further study.

In section 5, we introduced dataflow analysis techniques to minimize the spilling of address registers under static buffering. How useful and effective these techniques are depend both on the number of available address registers and on how expensive a spill is. For example, in the Motorola DSP56001, eight registers are available for addressing, while spills can often be performed with no runtime overhead (by doing them in parallel with other operations [25]). In contrast, in the MIPS R3000, any of the available 32 registers can be used for addressing, and at least one instruction cycle is required for a spill. Being able to accurately and efficiently estimate the effects of spilling would be useful in deciding between static and dynamic buffering.

The following section developed lifetime analysis techniques that aid in reducing storage requirements for buffers. An important area for further investigation is the incorporation of addressing tradeoffs between contiguous and scattered buffering. For example, if a logical buffer of length N is assigned to N mutually noncontiguous memory locations, then in general N absolute addresses must be employed. For programmable DSPs such as the DSP56001, arbitrary absolute addresses require an additional word of program memory and an additional instruction cycle, while register-indirect accesses to a contiguous buffer involve no program memory overhead and can often be performed in parallel with other operations. In contrast, many general purpose RISC processors allow large absolute displacements to be accessed through single-word instructions, but they do not allow register-indirect accesses to issue in parallel with other instructions. Furthermore, many do not support hardware autoincrement — a separate instruction must be issued to update the buffer pointer. Thus, more aggressive scattering of buffers may favor such RISC processors, while there is a strong trade-off between buffer storage, address storage, and execution time in the DSP56001.

Also, a scattered buffer can consist of multiple contiguous blocks of memory, each of which is accessed through a separate buffer pointer. Managing these multiple buffer pointers introduces another machine-dependent trade-off. Further examining the machine-dependent aspects of contiguous vs. scattered buffering is an important direction for future work.

Finally, we presented techniques to reduce modulo addressing overhead for both static and dynamic buffers. These techniques apply whenever modulo buffers are used, but how much improvement is gained depends on how expensive a modulo address update is in the target processor.

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.

References

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

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

[3] S. S. Bhattacharyya, E. A. Lee, "Memory Management for Synchronous Dataflow Programs", Memorandum UCB/ERL M92/128, Electronics Research Laboratory, University of California at Berkeley, November 1992.

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

[5] 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.

[6] 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.

[7] 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, 1975.

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

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

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

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

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

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

[14] 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.

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

[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.

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

[26] H. Printz, “Automatic Mapping of Large Signal Processing Systems to a Parallel Machine”, Memorandum CMU-CS-91-101, School of Computer Science, Carnegie Mellon University, May 1991.

[27] 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.

[28] G. Sih, “Multiprocessor Scheduling to Account for Interprocessor Communication,” Memorandum UCB/ERL M91/29, Electronics Research Laboratory, University of California at Berkeley, May 1991.

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