Shift Buffering Technique for Automatic Code Synthesis from Synchronous Dataflow Graphs

Hyunok Oh
CECS
University of California Irvine
CA, 92697, USA
+1 (949) 824-6725
oho@iris.snu.ac.kr

Nikil Dutt
CECS
University of California Irvine
CA, 92697, USA
+1 (949) 824-7219
dutt@ics.uci.edu

Soonhoi Ha
School of EECS
Seoul National University
Seoul, Korea
+82 (2) 880-7292
sha@iris.snu.ac.kr

ABSTRACT

This paper proposes a new efficient buffer management technique called shift buffering for automatic code synthesis from synchronous dataflow graphs (SDF). Two previous buffer management methods, linear buffering and modulo (or circular) buffering, assume that samples are queued in the arc buffers in the arrival order and are accessed by moving the buffer indices. But both methods have significant overhead for general multi-rate systems: the linear buffering method requires large size buffers and the modulo buffering method needs run-time overhead of buffer index computation. The proposed shift buffering method shifts samples rather than moving buffer indices. We develop optimal shift buffering algorithms to minimize the number of shifted samples. Our experimental results show that the proposed algorithm saves up to 90% of performance overhead while requiring the same amount of buffer memory as modulo buffering. Considering the sample copy overhead, shift buffering is applicable when memory size is more crucial than performance overhead, and the shifting overhead is less than the modulo addressing overhead. Another advantage of the shift buffering technique is that it supports the library code written with the linear buffering assumption, which is practically more important.

Categories and Subject Descriptors

C.3 [**SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS**]:Real-time and embedded systems – *real-time and embedded systems, Signal processing systems.*

General Terms

Algorithms, Languages.

Keywords

buffer management, automatic code synthesis, synchronous dataflow, shift buffering, modulo buffering

1. Introduction

As system complexity increases and fast design turn-around time becomes important, it attracts more attention to use high level software design methodology: automatic code generation from block diagram specification. COSSAP [1], GRAPE [2], and Ptolemy [3] are well-known design environments, especially for digital signal processing applications, with automatic code synthesis facility from graphical dataflow programs.

In a hierarchical dataflow program graph, a node, or a block, represents a function that transforms input data streams into output streams. The functionality of an atomic node is described in a high-level language such as C or VHDL. An arc represents a channel that carries streams of data samples from the source node to the destination node. The number of samples produced (or consumed) per node firing is called the output (or the input) sample rate of the node. In case the number of samples consumed or produced on each arc is statically determined and can be any integer, the graph is called a synchronous dataflow graph (SDF) [4] which is widely adopted in aforementioned design environments. We illustrate an example of SDF graph in Figure 1(a). Each arc is annotated with the number of samples consumed or produced per node execution.

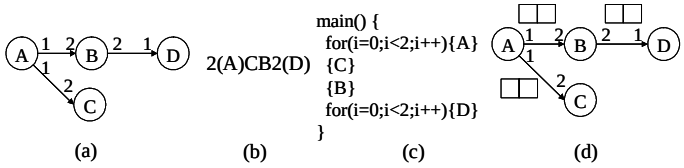


Figure 1. (a) SDF graph example, (b) a scheduling result, (c) a code template, and (d) the associated buffer allocation

To generate a code from the given SDF graph, the order of block executions is determined at compile time, which is called *scheduling*. Since a dataflow graph specifies only partial orders between blocks, there are usually more than one valid schedules. Figure 1(b) shows one of many possible scheduling results in a list form, where 2(A) means that block A is executed twice. The schedule will be repeated with the streams of input samples to the application. A code template according to the schedule of Figure 1(b) is shown in Figure 1(c).

When synthesizing software from an SDF graph, a buffer space is allocated to each arc to store the data samples between the source and the destination blocks. The number of allocated buffer entries should be no less than the maximum number of samples accumulated on the arc at run-time. After block A is executed twice, two data samples are produced on each output arc as explicitly drawn in Figure 1(d).

Since we assume that the kernel code of each function block of an SDF graph is given, the efficiency of data transfer between nodes

CODES+ISSS'05, Sept. 19–21, 2005, Jersey City, New Jersey, USA.

is the crucial performance factor of the automatically synthesized code from the graph. For a graph with large sample-rate changes, a naive buffering technique may require too large memory space to fit in the on-chip data memory. Therefore we are concerned with optimal buffer management in automatic code synthesis from SDF graphs in this paper.

Previous works have used three buffer management methods: static buffering, linear buffering, and modulo buffering. These methods assume that samples are queued in the arc buffers in the arrival order and are accessed by moving the buffer indices. But these methods are not optimal for general multi-rate systems. The static buffering is applicable only to a limited case that the buffer indices do not change at run time. Linear and modulo buffering methods have significant overhead for general multi-rate systems: the linear buffering method requires large size buffers and the modulo buffering method needs run-time overhead of buffer index computation. This paper proposes a new efficient buffer management technique called *shift buffering*. The shift buffering method shifts samples rather than moving buffer indices. We present optimal shift buffering algorithms to minimize the number of shifted samples.

The main contributions of this paper can be summarized as follows:

- It allows us to use the library block whose kernel code is written with linear buffering assumption.
- It supports code sharing between the nodes of the same function block by generating function style code.
- It makes an optimal tradeoff between performance and memory size.
- It reduces performance overhead compared with modulo buffering.

In section 2 we define some terminologies. We will discuss the previous buffer management techniques in section 3 and propose a new buffer management technique in section 4. In section 5, we explain optimal buffering algorithms and show experimental results in section 6. We make a conclusion in section 7.

2. Notations

We use the following notations to represent parameters of arc *a* in an SDF graph.

src(a) : the source node of arc *a* that produces samples.

sink(a) : the sink node of arc *a* that consumes samples.

p(a) : the number of samples produced per execution of *src(a).*

c(a) : the number of samples consumed per execution of *sink(a).*

d(a) : the number of initial delay samples on arc *a.*

bs(a) : the buffer size allocated on arc *a*.

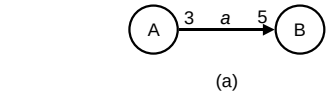
For the example of arc *AB* in Figure 1(d), *src(AB)* = A, *sink(AB)*=B, *p(AB)*=1, *c(AB)*=2, *d(AB)*=0, and *bs(AB)*=2.

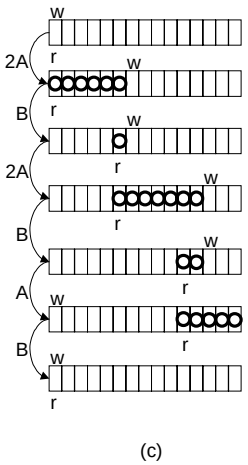
3. Previous Buffer Management Techniques

In this section, we explain three existent buffer management techniques: static buffering, linear buffering, and modulo (or circular) buffering [5] with a simple multi-rate example as shown in Figure 2.

We assume that the arc buffer is contiguous. For each contiguous buffer, we must determine whether modulo operation is necessary for accessing the buffer. If we should access the buffer with modulo operation then we call it modulo buffering; otherwise it is called linear buffering. The static buffering is a special case of the linear buffering where the buffer indices return to the original positions after iterating the static schedule. Apparently modulo operation requires performance overhead of modulo operation for each buffer access, especially for general purpose RISC microprocessors which do not support modulo addressing in hardware.

Consider Figure 2(a) with a schedule of Figure 2(b). The schedule is by no means optimal. But we use this schedule to reduce the buffer requirement with modulo addressing (Figure 3). Linear buffering requires that the minimum buffer size should be 15 that is the LCM of *p(a)* and *c(a).* As shown in Figure 2(c) the read and write indices can be safely incremented in the body of blocks A and B without reaching the buffer boundary. When node A is executed twice, 6 samples are produced and the write index becomes 6. When node B consumes 5 samples, the read index becomes 5. After an iteration of schedule, both the read index and the write index are reset to zero: So it is static buffering. Figure 2(d) represents the generated code template with linear buffering in which each buffer access code requires an addition operation to compute the memory address.






```
main()
{
  int a[15],r=0,w=0,i,j;
  for(;;) {
    for(i=0;i<2;i++) {
      for(j=0;j<2;j++) {
        // A's invocation.
        // access of a[w],a[w+1],a[w+2];
        w += 3;
      }
      // B' invocation
      // access of a[r],…,a[r+4];
      r += 5;
    }
    // A's invocation.
    // access of a[w],a[w+1],a[w+2];
    w = 0;
    // B' invocation
    // access of a[r],…,a[r+4];
    r=0;
  }
```

(d)

Figure 2. (a) An SDF graph, (b) its schedule, (c) read and write indices movement in linear buffering, and (d) the generated code.

Since the code for buffer access is simple and independent of the buffer size in linear buffering, a library block is usually written assuming linear buffering. If a library code is provided by a function prototype and is not liable to change, only linear buffering is allowed even though it requires large buffer memory.

But if there is an initial sample on the arc, linear buffering is not possible. Suppose that there is an initial sample on arc *a* in Figure 2. Then no buffer size is suitable for linear buffering. Only modulo buffering is able to handle the initial samples, to make it the most general buffering method of SDF graphs.

The modulo buffering method enables us to reduce the buffer size: the buffer size can decrease to 7 in Figure 2(a). But the buffer

access code should be changed in the block definition in the modulo buffering method: memory accesses should be wrapped-around when an index exceeds the buffer size. For instance, at the second invocation of node B in Figure 3(a), the values of the read index become {5,6,0,1,2}, and modulo operations are required to access the buffer in the order of {5,6,0,1,2}. Therefore the generate code has run time overhead of modulo operation for buffer index computation whenever the buffer is accessed as shown in Figure 3(b).

Note that if we want to eliminate modulo operations in the first invocation of node *B*, we should generate two different definitions of node B, with linear buffering and modulo buffering. But it is not a good solution since it is against the code reuse principle.

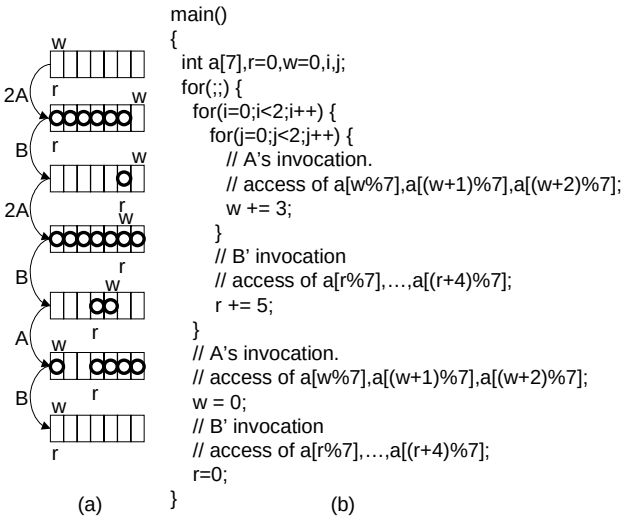


Figure 3. (a) Modulo buffering for both write and read indices and (b) the generated code.

In short, the modulo buffering is preferable to the memory-constrained systems while it costs non-negligible performance overhead of index computation. For example, in Figure 3(b) we need 30 modulo operations, which take 0.27 us on Pentium 4 3GHz for an iteration of schedule. Moreover it may not reuse the library blocks that are written assuming linear buffering without extra sample copy overhead.

4. Proposed Shift Buffer Management

We propose a new buffer management technique called shift buffering in order to reduce the performance overhead of the modulo buffering as well as to support library blocks written based on the linear buffering assumption. The shift buffering is different from the previous buffer management schemes since it shifts samples rather than moving index pointers.

In shift buffering, the requirement buffer size is equal to that in modulo buffering. For the example of Figure 2(a), Figure 4 shows a shift buffering in which some samples are shifted to keep an index from exceeding the buffer size. For instance, one sample is shifted after B is executed at the first invocation. Otherwise the write index would reach the buffer boundary during the next invocation of node A.

Note that the generated code of Figure 4(b) is similar to the linear buffering in Figure 2(d) except that the shift buffering copies data samples between block invocations. The performance overhead of shift buffering is the memory copy overhead of three samples for an iteration of the schedule, which takes 0.02 us on Pentium 4 3GHz that is much shorter than 0.27 us in the modulo buffering case.

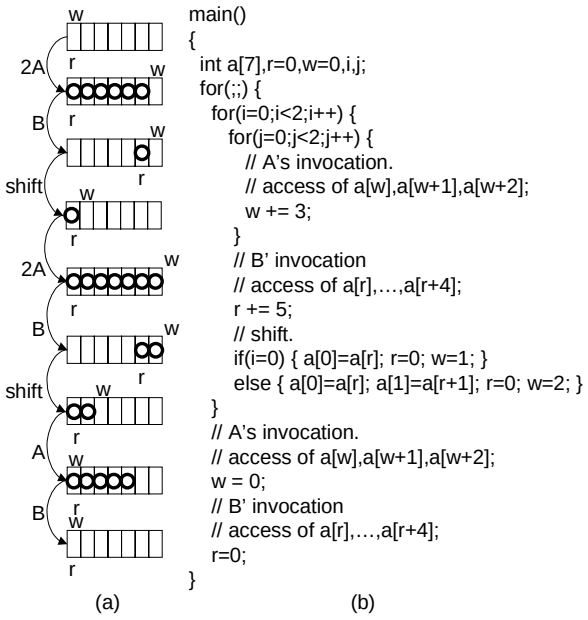


Figure 4. Shift buffering for Figure 2(a) and (b): (a) index movements and (b) the generated code with shift buffering.

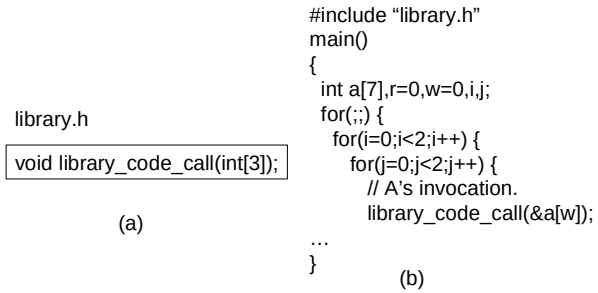
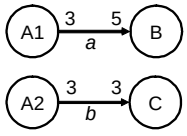


Figure 5. Importing a library: (a) a library header file and (b) node A calls a function in the library in Figure 4(b).

Since all buffer accesses inside the block use linear buffering, we can import a library code written with linear buffering assumption. For instance, assume that a library block is provided with functions declaration as header file as shown in Figure 5(a). Then the function arguments or input buffers should be contiguously accessible with linear buffering.

When a graph has multiple nodes associated with the same function block, function style shared code can be generated as shown in Figure 6. In the shared code generation, a single block definition should be provided as a function. Therefore the modulo buffering is not applicable for the block definition.

Thus the shift buffering is a novel buffer management technique to minimize the buffer size while preserving the linear buffering inside the block definition. It has benefits over modulo buffering when the memory copy overhead is less than the modulo operation overhead and the memory size is more critical than the performance overhead especially in multirate systems.



(a)

```
void A(int x[3]) { … }

main()
{
  int a[7],b[3],wa=0,wb=0,i,j;
  for(;;) {
…
     // A1's invocation
     A(&a[wa])
     wa += 3;
…
     // A2's invocation
     A(&b[wb])
     wb +=3;
…
}
```

(b)

Figure 6. Code sharing example: (a) a graph in which node A1 and A2 have common codes and (b) shared code.

The key question in shift buffering is when we have to shift the samples. Since it requires memory copy overhead to shift each sample, the optimal shift buffering problem is to minimize the number of shifted samples for an iteration of the schedule. It is a rather obvious observation on the shift position that we should shift samples after a sink node is fired and before a source node is executed. Let *A=src(a)* and *B=sink(a)* on arc *a*. Moreover a_k and b_k denotes the repetition counter (or looping number) of A and B respectively. Since we are interested in the buffer on arc *a*, we reduce the overall schedule to the schedule of block A and B omitting other blocks in the schedule. For example, if the schedule is *2ABCDAB*, where *C* and *D* are other blocks, then we reduce it to *2ABAB*. Then the shift position should lie between B and A, not between A and B. We summarize it as the following theorem.

Theorem 1. An optimal shift should be occurred before executions of *src(a)* and after *sink(a)* of an arc *a*.

By theorem 1, the optimum shift buffering problem is to determine the shift position between *B* and *A* in the schedule to minimize the number of shifted sample on the condition that indices accessing the buffer should not exceed the buffer size.

We can formalize the problem as follows: Let s_i be a binary variable in $a_1 Ab_1 Bs_1\ a_2 Ab_2 Bs_2\ \ldots a_k Ab_k Bs_k \ldots\ a_n Ab_n Bs_n$ where s_i =0 indicates that we do not shift samples and s_i =1 does we shift them. And let $w_k(a)$ be a write index accessed by the *kth* lexical appearance of *src(a)* and $r_k(a)$ a read index accessed by the *kth* of *sink(a)* in the schedule. Then the optimal shift buffering problem is to minimize $\sum_{i=1}^{n} e_i * s_i$ where e_i is the number of remained samples at shift position of s_i while $w_i(a) \le bs(a)$ for all *i*. Note that $e_k = \sum_{i=1}^{k} a_i * p(a) - b_i * c(a)$ and $r_i(a) \le w_i(a)$ since the read index may not be ahead of the write index.

5. Algorithm for Optimal Shift Buffering

In this section we present the algorithms for optimal shift buffering. First, we introduce an algorithm for an arc with no initial delay sample, of which the time complexity is $O(n^2)$. And we extend it for an arc with initial delay samples, of which the time complexity is $O(n^3)$.

5.1 Algorithm for an arc with no initial delay sample

Let $f(k) = e_k + \sum_{i=k+1}^{n} e_i * s_i$ to indicate the minimum number of shifted samples from s_k to s_n assuming that we shift samples at s_k, that is s_k=1 in *f(k)*. Then the *f(0)* represents the total number of shifted samples where e_0 is defined as 0.

In order to compute *f(k)*, we examine the write index *w(a)* that should not exceed the buffer size. When samples are shifted at s_k and are not from s_{k+1} to s_{k+m-1}, *w(a)* becomes $e_k + \sum_{i=k+1}^{k+m} a_i * p(a)$.

Suppose that $e_k + \sum_{i=k+1}^{k+m} a_i * p(a) \le bs(a) < e_k + \sum_{i=k+1}^{k+m+1} a_i * p(a)$. Then we should shift samples at least once between s_{k+1} and s_{k+m}. Since *f(k+h)* represents optimal shift result from s_{k+h} to s_n, *f(k)* chooses the minimum value among *f(k+1), f(k+2) ,…, f(k+m).*

Algorithm 1.

$f(n) = e_n = 0$.

$f(k) = e_k + \min(f(k+1), f(k+2), \ldots, f(k+m))$ where m is maximum while $w(a) = e_k + \sum_{i=k+1}^{k+m} a_i * p(a) \le bs(a)$.

Since *f(0)* represents the minimum number of shifted samples, we can find the optimal shift occurrences by retracing *f(0)*. Since *O(n)* time is required to compute *f(k)*, the time complexity of the given algorithm is $O(n^2)$.

Consider Figure 7 and assume that schedule is *2AB 6(AB)* and buffer size of *bs(a)* is 25. The required buffer size for the linear buffering is 56. The optimal shift buffering problem is to determine $s_1, \ldots, s_7$ in $2ABs_1ABs_2ABs_3ABs_4ABs_5ABs_6ABs_7$. Compute the number of remained samples at each shift position : $(e_1, e_2, e_3, e_4, e_5, e_6, e_7)$=(6,5,4,3,2,1,0).

Now let us compute *f(k)*.

$f(7) = e_7 = 0.\ \ f(6) = e_6 + f(7) = 1.$

$f(5) = e_5 + \min(f(6), f(7)) = 2,\ e_5 + 2 * p(a) = 16 \le 25.$

$f(4) = e_4 + \min(f(5), f(6), f(7)) = 3,\ e_4 + 3 * p(a) = 24 \le 25.$

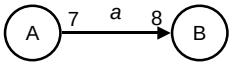
$f(3) = e_3 + \min(f(4), f(5), f(6)) = 5,\ e_3 + 3 * p(a) = 25 \le 25.$

$f(2) = e_2 + \min(f(3), f(4)) = 8,\ e_2 + 2 * p(a) = 19 \le 25.$

$f(1) = e_1 + \min(f(2), f(3)) = 11,\ e_1 + 2 * p(a) = 20 \le 25.$

$f(0) = e_0 + \min(f(1), f(2)) = 0 + 8,\ e_0 + 3 * p(a) = 21 \le 25.$

The minimum number of shifted samples is 8 and $s_2 = s_4 = s_7 = 1$ since *f(2), f(4)* and *f(7)* are used for computing *f(0)*. Therefore we make a schedule of *2AB AB (shift) 2(AB) (shift) 3(AB) (shift)*.



2AB 6(AB)

Buffer size = 25

Figure 7. An SDF graph without initial delay samples.

Note that we flatten the looped schedule when computing the optimal shift position and re-loop the modified schedule after inserting the shift operation. Thus the proposed technique restructures a given looped schedule with the reduced buffer size.

5.2 Algorithm for an arc with delay samples

When an arc has initial delay samples, we are not sure whether the optimal shift buffering result shifts samples at s_n. Therefore we assume that we shift samples at s_q. By combining *f(k)* with this s_q, we introduce a new function *f(k,q)* that indicates the minimum number of shifted samples when we shift samples at s_k and s_q but do not shift samples after s_q. Or let *f(k,q)* denote the minimum number of shifted samples where s_k =1, s_q =1, and s_{q+1} = s_{q+2} =…=s_n =0. We know simply that $f(k,q) = e_k + e_q + \sum_{i=k+1}^{q-1} e_i * s_i$ since $\sum_{i=k+1}^{q-1} e_i * s_i$ represents the minimum number of shifted samples from s_{k+1} to s_{q-1}. *f(k,q)* is defined when $k \le q$ and *f(k,k)* = e_k.

Similarly to algorithm 1, we compute *f(k,q)* by choosing the minimum value among *f(k+1,q)*, …, *f(k+m,q)* where $e_k + \sum_{i=k+1}^{k+m} a_i * p(a) \le bs(a) < e_k + \sum_{i=k+1}^{k+m+1} a_i * p(a)$.

The initial value of *w(a)* is dependent on *q* that is last shift position. If samples are shifted at s_q and are not shifted at $s_{q+1},\ldots,s_n$ then the initial value of *w(a)* is $e_q + \sum_{i=q+1}^{n} a_i * p(a)$.

Since *f(0,q)* is defined as min(*f(1,q),f(2,q),…,f(m,q)*) where $w(a) = e_q + \sum_{i=q+1}^{n} a_i * p(a) + \sum_{i=1}^{m} a_i * p(a) \le bs(a)$ we can find the optimal shift buffering positions by choosing the minimum *f(0,q)* among $0 \le q \le n$.

Algorithm 2.

f(k,k) = e_k.

f(k,q) = e_k + min(*f(k+1,q),f(k+2,q),…,f(k+m,q)*) where *m* is the maximum value satisfying $w(a) = e_k + \sum_{i=k+1}^{k+m} a_i * p(a) \le bs(a)$ and $k + m \le q$.

f(0,q) = min(*f(1,q),f(2,q),…,f(m,q)*) where *m* is the maximum value satisfying $w(a) = e_q + \sum_{i=q+1}^{n} a_i * p(a) + \sum_{i=1}^{m} a_i * p(a) \le bs(a)$.

$f(0) = \min_{q=1,\ldots,n} f(0,q)$

Since this algorithm requires *n* times more than the previous algorithm for *q*, the time complexity is $O(n^3)$.

Consider Figure 8 in which arc *a* has 4 initial delay samples. Assume that the schedule is *4(AB) 2AB 2(AB)* and the buffer size is 25. Then the number of shifted samples vector (e_1, …, e_7) is (3,2,1,0,6,5,4) and repetition vector for node *A* (a_1,…, a_7) = (1,1,1,1,2,1,1). Now we compute *f*(7,7), *f*(6,7), … , *f*(0,7).

$f(7,7) = e_7 = 4 \quad f(6,7) = e_6 + f(7,7) = 9$

$f(5,7) = e_5 + \min(f(6,7), f(7,7)) = 6 + f(7,7) = 10$

$e_5 + (a_6 + a_7) * p(a) = 6 + 2 * 7 = 20 \le 25$

$f(4,7) = e_4 + \min(f(5,7), f(6,7)) = 0 + f(6,7) = 9$

$e_4 + (a_5 + a_6) * p(a) = 0 + 3 * 7 = 21 \le 25$

$f(3,7) = e_3 + \min(f(4,7), f(5,7)) = 1 + f(4,7) = 10$

$e_3 + (a_4 + a_5) * p(a) = 1 + 3 * 7 = 22 \le 25$

$f(2,7) = e_2 + \min(f(3,7), f(4,7)) = 2 + f(4,7) = 11$

$e_2 + (a_3 + a_4) * p(a) = 2 + 2 * 7 = 16 \le 25$

$f(1,7) = e_1 + \min(f(2,7), f(3,7), f(4,7)) = 3 + f(4,7) = 12$

$e_1 + (a_2 + a_3 + a_4) * p(a) = 3 + 3 * 7 = 24 \le 25$

$f(0,7) = \min(f(1,7), f(2,7), f(3,7)) = 10$

$e_7 + (a_1 + a_2 + a_3) * p(a) = 4 + 3 * 7 = 25 \le 25$

Similarly we can compute *f*(*k*,6) values shown as the third column in Table 1. After computing all *f*(0,*q*) values like the last row in Table 1, we choose the minimum value among *f*(0,*q*), which is *f*(0,6) (= 8). Hence the total number of shifted samples is 8. Moreover shift positions are s_1, s_4 and s_6 since *f*(0,6) is computed by *f*(1,6), *f*(4,6) and *f*(6,6).

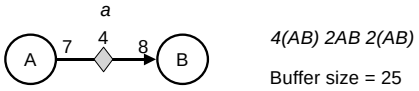


Figure 8. An SDF graph with initial delay samples.

Table 1. f(k,q) computation of Figure 8.

k \ q	7	6	5	4	3	2	1
7	4						
6	9	**5**					
5	10	11	6				
4	9	**5**	6	0			
3	10	6	7	1	1		
2	11	7	8	3	3	2	
1	12	**8**	9	4	4	5	3
0	10	**8**	N/A	N/A	N/A	N/A	N/A

Although we can prove the algorithm is optimal, we skip the proof due to the space limitation.

Theorem 2. If *d(a)* is no larger than *p(a)+c(a)-g(a)* on arc *a*, then the lower bound of buffer size is *p(a)+c(a)-g(a)+md(a)*. Then the minimum number of shifted samples is $g(a) * \frac{m(a) * (m(a) - 1)}{2} + m(a) * md(a)$ **where *g(a)*=g.c.d.(*p(a),c(a)*), *m(a)*=min(*p(a), c(a)*)/ *g(a)* and *md(a)* = *d(a)* mod *g(a)*.**

For the example of Figure 8, the optimal buffer size is 14(=7+8-1+(4 mod 1)) and the minimum number of shifted samples is 21(=1*7*6/2+7*(4 mod 1)).

Unfortunately, if the number of delay samples is larger than *p(a)+c(a)*-gcd(*p(a),c(a)*) then it is hard to make a formulation since it does not guarantee that at every shift position samples are shifted.

6. Experiments

In this section, we show the tradeoff between performance overhead and buffer size in our shift buffering and compare the shift buffering algorithm with modulo buffering in terms of performance overhead. Comparison with linear buffering in terms of buffer size is too obvious to be repeated in the experiments.

First we examined the tradeoff between the number of instruction and the buffer size in the example of Figure 8. Figure 9 represents the tradeoff where x axis indicates the buffer size and y axis does the number of shifted samples. The number of shifted samples is

inversely proportional to some power of buffer size when the graph is curve-fitted.

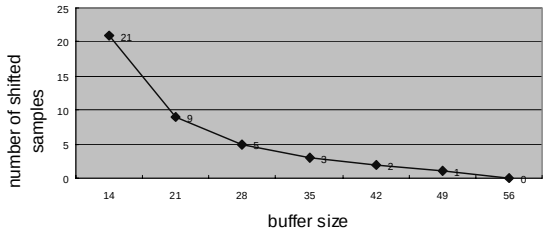


Figure 9. Tradeoff between buffer size and the number of shifted samples in Figure 8

When we compare shift buffering with modulo buffering we use a simple conditional execution "(x<M?x:x-M)" replacing a modulo operation "(x%M)" since the conditional execution is about three times as fast as the modulo operation.

Figure 10 illustrates a CD2DAT application that converts CD format (44.1 KHz sampling data) to DAT format (48 KHz). Each arc requires 4, 10, and 11 size buffers at least to hold the live samples while each functional node has additional buffers since it need to store the previous samples. The repetitions counts for FIR1, FIR2, FIR3 and FIR4 are respectively 147, 98, 56 and 40 invocations per period.

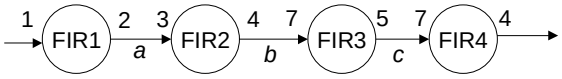


Figure 10. A CD2DAT algorithm

In this application, the total number of shifted samples is 213. On arc *a*, 49 samples are shifted since *g(a)*m(a)*(m(a)-1)/2=1*2*1/2=1* and (2FIR1)(FIR2)(FIR1)(FIR2) are called 49 times. On arc *b*, 84 samples since 1*4*3/2=6 and (3((2FIR2)(FIR3)))(FIR2)(FIR3) are called 14 times. On arc *c*, 80 samples since 1*5*4/2=10 and (2((2FIR3)(FIR4)(FIR3)(FIR4)))(FIR3)(FIR4) are called 8 times.

On the other hand, 1638 modulo operations are required for modulo buffering. The read index of FIR2 needs 294 mod operations and the write index requires 392 mod operations since the number of invocations of FIR2 is 98 times and at each time it reads 3 samples and writes 4 samples. The read index and the write index of FIR3 need 392 and 280 mod operations respectively due to 56 invocations of FIR3. The read index of FIR4 requires 280 mod operations due to 40 invocations of FIR4.

Another example is a non-uniform filter bank in which the low pass filters retain 2/3 of the spectrum while the high pass filters retain 1/3 [7].

Table 2. Performance overhead improvements of shift buffering over modulo buffering

	ARM720T	ARM920T	XScale	P4
CD2DAT	42.4%	59.08%	66.62%	61.11%
Filter bank	78.78%	79.17%	81.32%	90.45%

Table 2 summarizes the performance overhead improvement of shift buffering over modulo buffering, which is computed by (modulo buffering time-shift buffering time)/(modulo buffering time). We have measured buffering execution time on ARM720T, ARM920T, XScale and Pentium4 3GHz. We reduce the performance overhead of buffer management up to 90% by using shift buffering.

7. Conclusions

In this paper, we propose a new buffer management algorithm called shift buffering. The shift buffering is motivated to overcome the performance overhead of the modulo buffering. In order to minimize the number of shifted samples, we propose optimal algorithms to find the optimal shift positions for an arc without and with initial delay samples in $O(n^2)$ and $O(n^3)$ time complexities respectively. The proposed shift buffering provides a unique implementation possibility where the library block is written in linear buffering and there is an initial delay samples on the arc.

The experimental result shows that the proposed shift buffering algorithm reduces up to 90% of the buffer access overhead compared with modulo buffering algorithm. In the future, we will extend it to hardware synthesis in which shift buffering is more beneficial in terms of hardware area and power consumption as well as cycle overhead.

Acknowledgement

This work was partially supported by NSF grants CCR-0203813, ACI-0204028, National Research Laboratory Program (Grant No. M1-0104-00-0015), and IT leading R&D Support Project funded by Korean MIC.

References

[1] Synopsys Inc., 700 E. Middlefield Rd., Mountain View, CA94043, USA, COSSAP User's Manual.

[2] R. Lauwereins, M. Engels, J. A. Peperstraete, E. Steegmans, and J. Van Ginderdeuren, "GRAPE: A CASE Tool for Digital Signal Parallel Processing", IEEE ASSP Magazine, vol. 7, (no.2):32-43, April, 1990

[3] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschimitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems", Int. Journal of Computer Simulation, special issue on "Simulation Software Development", vol. 4, pp. 155-182, April, 1994.

[4] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous dataflow programs for digital signal processing," *IEEE Trans. Comput.*, vol. C-36, no. 1, pp. 24-35, Jan 1987.

[5] S. S. Bhattacharyya and E. A. Lee, "Memory management for dataflow programming of multirate signal processing algorithms," *IEEE Trans. Signal Processing*, vol. 42, No. 5, May, 1994.

[6] S. S. Bhattacharyya, P.K. Murthy and E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, Norwell Ma, 1996.

[7] P. K. Murthy, S. S. Bhattacharyya and E. A. Lee, "Joint Minimization of Code and Data for Synchronous Dataflow Programs," *In the Journal of Formal Methods in System Design*, vol. 11, no. 1, July, 1997.