Single Appearance Schedule with Dynamic Loop Count for Minimum Data Buffer from Synchronous Dataflow Graphs

Hyunok Oh
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697, USA
oho@iris.snu.ac.kr

Nikil Dutt
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697, USA
dutt@ics.uci.edu

Soonhoi Ha
School of EECS
Seoul National University
Seoul,151-742,Korea
sha@iris.snu.ac.kr

ABSTRACT

In this paper, we propose a new single appearance schedule for synchronous dataflow programs to minimize data memory and code memory size at the same time. When the software code is automatically synthesized from the dataflow program graphs, a single appearance schedule promises only one appearance of each node definition in the generated code. While several heuristics have been developed to find a single appearance schedule, they all have to pay significant amount of data memory overhead compared with a buffer optimal schedule. The key idea of the proposed technique is to make a dynamic decision of loop count to make a schedule *quasi-static*. The proposed quasi-static static schedule produces a single appearance schedule code with minimum data memory requirement. We prove that the proposed scheduling technique is optimal for a chain-structured graph in terms of data memory requirement while maintaining the single appearance schedule. The only penalty for the proposed technique is slight performance overhead of computing loop counts dynamically. Experimental results show that the proposed algorithm reduces 20% total memory with less than 1% performance overhead compared with the previous single appearance schedule algorithms for CD2DAT and non uniform filter bank applications.

Categories and Subject Descriptors

D.3.2 [**Language Classifications**]: Data-flow language

General Terms

Algorithm, Languages

Keywords

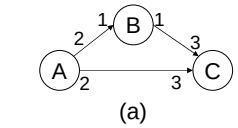
Synchronous Dataflow, Single Appearance Schedule, Memory Optimization,Dynamic Loop Count,Automatic Code Synthesis

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'05, September 24–27, 2005, San Francisco, California, USA.
Copyright 2005 ACM 1-59593-149-X/05/0009 ...$5.00.

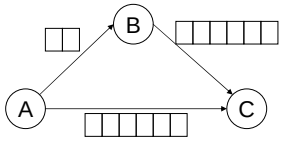
1. INTRODUCTION

As system complexity increases and fast design turn-around time becomes important, high level software design methodologies become critical. In the context of DSP applications, there have been several approaches to automatic code generation from block diagram specification including COSSAP [1], GRAPE [10], and Ptolemy [7]. It is also the main concern of this paper.

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) [11] 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.



(a)



(c)

```
3(A2B)2C
```
(b)

```
main()
{
 int i,j,k;
 for(;;) {
   for(i=0;i<3;i++) {
     // A's code
     for(j=0;j<2;j++) {
       // B's code
     }}
   for(k=0;k<2;k++){
     // C's code
}}}
```
(d)

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

To generate a code from the given SDF graph, the order of block executions is determined at compile time by static *scheduling* of the graph. Since a dataflow graph specifies only partial orders between blocks, there are usually several

valid schedules that satisfy the partial ordering. Figure 1(b) shows one of many possible scheduling results in a list form, where 2C means that node C 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(d). The block definition is inlined in the generated code, it is called inline-style. When a software code is automatically synthesized from an SDF graph, buffer space is allocated to each arc to store the data samples between the source and the destination blocks as shown in Figure 1 (c). The total buffer size becomes 14 in this example. The number of allocated buffer entries should be no less than the maximum number of samples accumulated on the arc at run-time.

If a schedule contains only one lexical appearance of each node, this schedule is called a single appearance schedule(SAS) (e.g. as 3(A2B)2C in Figure 1(b)). A single appearance schedule minimizes the code memory size since each block has a single definition in a generated code. Consider another schedule that is a non single appearance schedule, 2(A2B)C(A2B). Then, the generated code has two instances for nodes A and B while it reduces the data buffer size from 14 to 10 in Figure 1. In general while an SAS is preferable to minimize the code memory size, it requires larger buffer memory than a non SAS. The buffer size on each arc in a SAS is no less than the least common multiplier of the producing sample rate and consuming sample rate for the arc.

In this paper we propose a novel single appearance scheduling technique whose key idea is introducing a dynamic decision of loop count to make a schedule *quasi-static*. The proposed quasi-static schedule produces a single appearance schedule code with minimum data memory requirement. Section 2 defines some notations and section 3 reviews the related works. In section 4, we introduce motivational examples. The proposed technique is explained in section 5. We will show experimental results in section 6 and make a conclusion in section 7.

2. TERMINOLOGY

We use the following notation to represent the parameters of arc a and node v in SDF graphs.

$src(a)$: the source node of a that produces samples on the arc

$sink(a)$: the sink node of a that consumes samples from the arc

$p(a)$: the number of samples produced by an invocation of $src(a)$

$c(a)$: the number of samples consumed by an invocation of $sink(a)$

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

$inv(v)$: the total number of invocations of node v per period.

For arc AB in Figure 1, $src(AB) = A$, $sink(AB)=B$, $p(AB)=2$, $c(AB)=1$, $d(AB)=0$, $inv(A)=3$, $inv(B)=6$ and $inv(C)=2$.

3. RELATED WORKS

Since minimization of memory requirements in embedded system is crucial, many researches have been performed to find a schedule to minimize data memory and/or code memory.

Ade et al. [3, 4] have developed the formula on the upper bounds on the minimum buffer memory requirement for a number of restricted subclasses of delayless, acyclic graphs, including arbitrary-length chain-structured graphs. Some of these bounds have been generalized to handle delays in [5] which has shown that the problem of constructing a schedule that minimizes the buffer requirement is NP-complete.

Ritz et al. [15, 16] have proposed a buffer sharing optimization among a subset of single appearance schedules, called flat single appearance schedule. Since the flat SAS does not allow nested loops, it usually requires large buffer memory even though it shares buffers allocated on each arc. Murthy et al. have developed several heuristics that produce SAS with nested loop: APGAN, RPMC, and GDPPO [6, 13]. These algorithms have an inherent limitation that they require at least buffer memory of $LCM(p(a), c(a))$ for each arc a.

To overcome the limitation of SAS, some techniques have been developed, which give up the single appearance constraint for overall memory saving [17, 18, 8, 9]. These approaches observe the trade-off of code and data memory size and try to minimize the code memory overhead by generating function-style codes instead of inline-style code. By defining each block as a function call, a generated code from a non SAS has only one definition of each block but paying the extra overhead of function calls.

Buffer sharing algorithms [12, 14] have been proposed to minimize data memory. These sharing algorithms analyze buffer life time and share buffers of which life-times are not overlapped with each other.

The proposed technique is unique and novel that it minimizes the data memory while preserving the single appearance of block definition in the inline-style code.

4. MOTIVATION

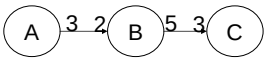
As discussed earlier, single appearance scheduling algorithms pay huge penalty of data memory for a graph with large sample rate changes. Moreover, no SAS exists for cyclic graphs in general. The following two examples show these limitations of SAS. With those examples we will introduce the proposed scheduling technique.

The first example is shown in Figure 2(a). The previous SAS algorithms produce 2A3B5C as the schedule result, which requires 6 and 15 data buffers on arc AB and arc BC respectively. If a buffer optimal non SAS algorithm is applied, the schedule becomes ABCABCCBCC (= 2(ABC) CB2C) requiring 4 and 7 data buffers, which is minimum buffer size while additional code memory is necessary to represent multiple appearances of node B and C in a code.

To avoid the multiple lexical appearances of nodes in buffer optimal non SAS we propose a dynamic loop count single appearance scheduling called dlcSAS which converts a buffer optimal non SAS result to a single appearance schedule while preserving the minimum buffer size. Examine the non SAS in Figure 2 (b). In the buffer optimal schedule, whenever a sink node has enough samples on its input arc it should be executed. Hence, node B can be executed twice after the second invocation of node A while node B can be executed only once after the first invocation of node A. In the proposed dlcSAS, we notate this varying loop count of node B as 2(A{1,2}B) meaning that the loop count values of node B are 1 and 2 alternatively every invocation of node A. We can compute the loop count values of node B by dividing the

number of accumulated samples by the input sample rate of node B. For instance, in Figure 2 (c), the loop count for node B ($= ka$) is the number of samples on arc AB ($= ra+3$) over the input rate of node B ($= 2$), which is $ka = (ra + 3)/2$. Note that ra indicates the number of samples on arc AB after node B is fired and before node A is executed. Therefore, the number of accumulated samples ($= ra$) is added by the produced number and subtracted by the consumed number.

Similarly, node C can be executed twice after the second and the third invocations of node B while it can be executed only once after the first invocation of node B. The schedule is represented as 3(B{1,2,2}C) in the proposed dlcSAS. By combining the two schedules, we obtain the final dlcSAS, 2(A{1,2}(B{1,2,2}C)). The generated code template from this dlcSAS is shown in Figure 2 (c). Note that the generated code has a single appearance of each block while preserving the minimum buffer memory as the buffer optimal non SAS. The generated code requires run time overhead of determining the loop count dynamically. However this performance overhead becomes negligible when the code size of each node becomes large.



(a)

SAS : 2A3B5C
Buffer-optimal non SAS :
ABCABCCBCC
dlcSAS :
2(A {1,2}(B {1,2,2}C))

(b)

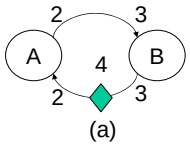
```
main()
{
 int n,i,j, a[4],b[7],ra=0,ka,rb=0,kb;
 for(;;) {
  for(n=0;n<2;n++) {
    /* A's code */
    ka=(ra+3)/2; ra += 3-2*ka;
    for(i=0;i<ka;i++) {
     /* B's code */
     kb=(rb+5)/3; rb += 5-3*kb;
     for(j=0;j<kb;j++) {
       /* C's code */
}}}}}
```

(c)

Figure 2: (a) An SDF graph (b) schedule results and (c) generated code by dlcSAS

The second example illustrates a cyclic graph that has no valid SAS as shown in Figure 3 where there are 4 initial delay samples on arc BA. 2ABAB is the only valid schedule and it is a buffer optimal non SAS. We can translate it as a dlcSAS that is 2({2,1}A B). It means that the first loop count of node A is 2 and the second is 1.

For the simple examples discussed above, dlcSAS may be regarded as a different representation of a buffer optimal non SAS. Every non SAS can be transformed into a dlcSAS by storing for each node loop count values as many as schedule length and setting 1 to loop count of a node when the node appears in non SAS. For instance, when ABAC schedule is given, four loop count values are assigned to every node A, B, and C. And then since node A is executed first and third, {1,0,1,0} is assigned. Similarly {0,1,0,0} and {0,0,0,1} are assigned to node B and C respectively. Therefore we can build the equivalent dlcSAS of 4({1,0,1,0}A {0,1,0,0}B {0,0,0,1}C). We are, however, interested in simple expression of loop count computation to minimize code memory and performance overhead. Therefore if we restrict the dlcSAS to contain only simple computation then a buffer optimal non SAS may not be represented as a dlcSAS since if we can represent it as a dlcSAS with simple expression then we can find a buffer optimal schedule by computing the expression, while finding a buffer optimal schedule is known as NP problem [13].



(a)

SAS : N/A
Buffer optimal non SAS :
2ABAB
dlcSAS : 2({2,1}A B)

(b)

```
main()
{
 int i,j, a[4],b[4],r=0,h;
 for(;;) {
  for(i=0;i<2;i++) {
    if(r>=1) {h=1;r-=1;}
    else {h=2;r+=1;}
    for(j=0;j<h;j++) {
      // A's code
    }
    // B's code
}}}
```

(c)

Figure 3: (a) An SDF graph with delay samples (b) schedule results and (c) generated code by dlcSAS

5. DYNAMIC LOOP COUNT SINGLE APPEARANCE SCHEDULING ALGORITHM

5.1 Dynamic Loop Count for Two Nodes

In this section, we first explain how to compute the dynamically varying count values and generate a code for the variation. For a pair of nodes, we can change the loop count value for a source node or a sink node to represent a buffer optimal schedule.

When we change the loop count value of the source node, the source node should be executed multiple times to produce the sufficient number of samples for the sink node. Let the accumulated number of samples on the arc as r. Then the loop count h of the source node should be no less than $\lceil \frac{c(a)-r}{p(a)} \rceil$. It means that the number of samples on the arc after the source node execution should be no less than $c(a)$. After executing both nodes, the accumulated number r is updated as $r+h*p(a)-c(a)$. For the minimum buffer requirement, the loop count of the source node is set to that bound as denoted as ${}_{p(a)}h_{c(a)}$ or ${}_{p(a)}h^{d(a)}_{c(a)}$ when $d(a) > 0$. The dlcSAS of Figure 3 belongs to this case.

When we change the loop count value of the sink node, the sink node can be executed until the accumulated samples are exhausted. Since there are $r + p(a)$ samples after execution of the source node and the sink node consumes $c(a)$ samples per execution, the sink node can be executed no more than $\lfloor \frac{r+p(a)}{c(a)} \rfloor$ times. Let k be the loop count of the sink node. After executing both nodes, there are $r + p(a) - k * c(a)$ samples on the arc. For the minimum buffer requirement, the loop count of the sink node is set to that bound as denoted as ${}_{p(a)}k_{c(a)}$ or ${}_{p(a)}k^{d(a)}_{c(a)}$ when $d(a) > 0$. The dlcSAS of Figure 2 belongs to this case. Note that users are free to choose any representation between loop count for source node and sink node.

Equation 1 summarizes the formulation of dynamic loop count in both cases.

EQUATION 1. *For each arc a, $r = d(a)$ initially and (i) if a schedule is $(h\,src(a))(sink(a))$, $h = \lceil \frac{c(a)-r}{p(a)} \rceil$ and $r = r + h * p(a) - c(a)$.*

(ii) if a schedule is $(src(a))(k\,sink(a))$, $k = \lfloor \frac{p(a)+r}{c(a)} \rfloor$ *and* $r = r + p(a) - k * c(a)$.

5.2 Optimization of Dynamic Loop Count Computation

The computation of dynamic loop count value as shown in Equation 1 requires ceiling or floor function. To avoid the ceiling or floor function, we devise another equation for the loop count computation as Equation 2.

EQUATION 2. *Let h denote a dynamic loop count variable for a source node, k for a sink node , and r the number of accumulated samples on arc a. The initial value of r is $d(a)$.*

(i) For the schedule of $(h\,src(a))(sink(a))$,

if $r \geq c(a) - (n-1) * p(a)$ *then* $h = n-1$ *and* $r = r + (n-1) * p(a) - c(a)$*; otherwise* $h = n$ *and* $r = r + n * p(a) - c(a)$ *where* $n = \lceil \frac{c(a)}{p(a)} \rceil$.

(ii) For the schedule of $(src(a))(k\,sink(a))$,

if $r \geq (n+1) * c(a) - p(a)$ *then* $k = n+1$ *and* $r = r + p(a) - (n+1) * c(a)$*; otherwise* $k = n$ *and* $r = r + p(a) - k * c(a)$ *where* $n = \lfloor \frac{p(a)}{c(a)} \rfloor$.

PROOF. Equation 2 comes from Equation 1 . Consider the case (i). At most $h * p(a) - c(a)$ samples are accumulated every sink node execution. If $r \geq c(a) - (n-1) * p(a)$ then $h = \lceil \frac{c(a)-r}{p(a)} \rceil \leq \lceil \frac{c(a)-c(a)+(n-1)*p(a)}{p(a)} \rceil = n-1$, which decreases r since $(n-1) * p(a) - c(a) = p(a) * (\lceil \frac{c(a)}{p(a)} \rceil - \frac{c(a)}{p(a)} - 1) < 0$. Therefore $r < (c(a) - (n-1) * p(a) + (n * p(a) - c(a)) = p(a)$ and $h = \lceil \frac{c(a)-r}{p(a)} \rceil \geq \lceil \frac{c(a)-p(a)}{p(a)} \rceil \geq \lceil \frac{c(a)}{p(a)} \rceil - 1 = n-1$. Consequently if $r \geq c(a) - (n-1) * p(a)$ then $h = n-1$; otherwise $h = n$ since $h * p(a) + r \geq c(a)$, $c(a) - h * p(a) \leq r < c(a) - (n-1) * p(a)$ and $h > n-1$.

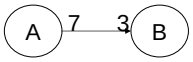
Consider the case (ii). If $r \geq (n+1) * c(a) - p(a)$ then $k = \lfloor \frac{p(a)+r}{c(a)} \rfloor \geq \lfloor \frac{p(a)+(n+1)*c(a)-p(a)}{c(a)} \rfloor = n+1$. Since $r < c(a)$, $k \leq n+1$. Therefore if $r \geq (n+1) * c(a) - p(a)$ then $k = n+1$; otherwise $k = n$ since $p(a) + r \geq k * c(a)$, $k * c(a) - p(a) \leq r < (n+1) * c(a) - p(a)$ and $k < n+1$. □

Even though Equation 2 looks more complex than Equation 1, it is actually simpler since $p(a)$, $c(a)$ and n are constant values. For instance, when $p(a) = 3$ and $c(a) = 7$, the generated code of the loop count computation becomes "if(r ≥ 1) { h=2; r-=1;} else {h=3;r+=2;}" by Equation 2. On the other hand, the code would be "h = ceil((7-r)/3); r += 3*h-7;" by Equation 1. The code by Equation 2 consists of a comparison, an assignment, and an arithmetic operation while the code by Equation 1 needs an assignment, 5 arithmetic operations, and a ceiling function.

Further optimization can be applied when a loop count value is 0 or 1. Consider Figure 4. When the loop count for a source node is changed, the generated code is represented like Figure 4 (b). Since the dlcSAS is ({1,0,1,0,1,0,0}A)B, the loop count is dynamically computed every execution of node B. However, the computation overhead can be reduced further as shown in Figure 4 (c). where the "for" loop is replaced by an "if" statement and the computation of loop count value becomes simpler.

The code template of this optimization is shown in Equation 3.

EQUATION 3. *(i) A code for a dynamic loop count for a source node when* $p(a) > c(a)$*:*



(a)

```
main()
{
 int i,j, a[9],r=0,h;
 for(;;) {
  if(r>=3) { h=0;r-=3;}
  else {h=1; r+=4;}
  for(i=0;i<h;i++) {
    // A's code
  }
  // B's code
 }}
```

(b)

```
main()
{
 int i,j, a[9],r=0,h=0;
 for(;;) {
  if(h==0) {
    // A's code
    if(r>=2) { h=2;r-=2;}
    else {h=1; r+=1;}
  } h--;
  // B's code
 }}
```

(c)

Figure 4: (a) An SDF graph (b) Unoptimized code (b) optimized code

int h=0, r=0;
...
if(h==0) {
// source node's code
*if(r ≥ (n+1)*c(a)-p(a)) {h=n;r-=(n+1)*c(a)-p(a);}*
*else {h=n-1; r+=p(a)-n*c(a);}*
}else h--;

// sink node's code

where $n = \lfloor \frac{p(a)}{c(a)} \rfloor$.

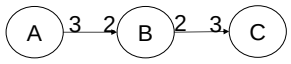
(ii) A code for a dynamic loop count for a sink node when $p(a) < c(a)$*:*

int k=n-1,r=0;
...
// A source node's code
if(k==0) {
// A sink node's code
*if(r ≥c(a)-(n-1)*p(a)) {k=n-2;r-=c(a)-(n-1)*p(a);}*
*else k=n-1;r+=n*p(a)-c(a);*
} else k--;

where $n = \lceil \frac{c(a)}{p(a)} \rceil$.

This optimization is useful when we should apply a dynamic count for a source node when $p(a) > c(a)$ or a dynamic count for a sink node when $p(a) < c(a)$. Consider Figure 5. If we want to avoid zero loop count value, we can make the dlcSAS between node A and B as "A{1,2}B" and "{2,1}BC" between B and C. Unfortunately, the two loop counts for node B are conflicting each other since {1,2}B and {2,1}B cannot be expressed simultaneously. Therefore the proposed dlcSAS scheduling algorithm generates "A{1,2}(B{0,1,1}C)". For the loop count computation of node C, Equation 3 can be used to obtain the code shown in Figure 5 (c).

If $p(a)$ divides $c(a)$ or $c(a)$ does $p(a)$ then we can generate more compact code as shown in Equation 4, since we do not need to update r which indicates the number of accumulated samples.



(a)

SAS : 2A3B2C
Buffer-optimal non SAS :
2(AB)CBC
dlcSAS : 2(A{1,2}(B{0,1,1}C))

(b)

```
main()
{
 int n,i,j, a[4],b[4],ra=0,ka,rb=0,
   kb=1;
 for(;;) {
  for(n=0;n<2;n++) {
   for(i=0;i<3;i++) {
     // A's code
     if(ra>=1) {ka=2;ra-=1;}
     else {ka=1;ra+=1;}
     for(j=0;j<ka;j++) {
       // B's code
       if(kb==0) {
         // C's code
         if(rb>=1) {kb=1; rb-=1;}
         else {kb=0; rb+=1;}
       } else kb--;
}}}}}
```

(c)

Figure 5: (a) An SDF graph (b) schedule results and (c) generated code by dlcSAS

EQUATION 4. *(i) A code for a loop count for a source node when $p(a) > c(a)$ and $c(a)$ divides $p(a)$:*

```
int h=0;
...
if(h==0) {
  // A source node's code
  h = n-1;
} else h--;
// A sink node's code
```

where $n = \frac{p(a)}{c(a)}$.

(ii) A code for a dynamic loop count for a sink node when $p(a) < c(a)$ and $p(a)$ divides $c(a)$:

```
int k=n-1;
...
// A source node's code
if(k==0) {
  // A sink node's code
  k = n-1;
} else k--;
```

where $n = \frac{c(a)}{p(a)}$.

5.3 Dynamic Loop Count for Chained Structure SDF Graphs

So far we have explained how to construct a dlcSAS between two nodes. In this section, we explain how to construct a dlcSAS for the entire graph. First we will explain a scheduling algorithm for a chain-structured graph as shown in Figure 6. We can cluster nodes in this graph in various ways such as A((BC)D), (AB)(CD), and so on. For example, the APGAN SAS algorithm clusters the graph to A((BC)D) and generates the schedule as 40A3(5(2B3C)3D) which requires 141 data buffers. However, the cluster of A((BC)D) is not feasible for dlcSAS. Here is the reason. First we have to make a dlcSAS for (BC) cluster. Assume that the loop count of node C is dynamic. Then the schedule becomes (B $_3k_2$ C). Now we have to make the next dlcSAS for the cluster of ((B $_3k_2$ C) D). Since the number of samples produced from (BC) cluster is not constant, it is not possible to determine the dynamic loop count of either node D nor (BC) cluster. Equations for the dynamic loop count computation assume that the production and the consumption rates are constant. If we vary the loop count of node C for the dlcSAS of (BC) cluster, the dlcSAS of (BC) cluster becomes ($_3h_2$B C). In this case, it is not possible to determine the dlcSAS between node A and cluster ((BC)D) since the consumption rate of node B varies dynamically.

Therefore, we propose two clustering algorithms for the dlcSAS construction of the entire graph: source node first (shortly source-dlcSAS) and sink node first (shortly sink-dlcSAS). For the graph of Figure 6, we cluster (((A)B)C)D and A(B(C(D))) by source-dlcSAS and sink-dlcSAS respectively. We assign a dynamic loop count to the source node by the source node first clustering algorithm or to the sink node by the sink node first algorithm.

If we cluster nodes by the source node first, then we build a code with a dynamic count for a source node, which is $5(_3h_2((_3h_4A)B)C)D$; otherwise a code with a count for a sink node, $(A_3k_4(B_3k_2(C_1k_5D))))$. In the former schedule, node A is executed $_3h_4$ times to produce the enough number of samples for node B to be executed. Node B is also executed $_3h_2$ times to produce the enough number of samples for one invocation of node C. Then, Node C is executed five time every invocation of node D. Therefore the schedule requires minimum data buffer size. We know that the later schedule also requires minimum data buffer size. Generally the dlcSAS algorithm produces buffer optimal code for chained structure graphs as stated in Theorem 1 below.

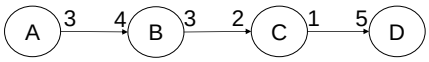


Figure 6: A chained structure SDF graph

LEMMA 1. *The schedule of $(h\, src(a))sink(a)$ requires minimum buffer memory on arc a for a graph with two nodes $src(a)$ and $sink(a)$.*

PROOF. We prove it by showing that the accumulated number of samples on the arc is always no more than the bound, $max(d(a), c(a)+p(a)-g+(d(a) \mod g))$ as reported in [4] where g denotes $gcd(p(a), c(a))$. By Equation 1, the number of accumulated samples is $r + h * p(a)$ where $h = \lceil \frac{c(a)-r}{p(a)} \rceil$. Let $c(a) - r = n * p(a) + m (0 \leq m < p(a))$.

If $m = 0$ then $h = \lceil \frac{n*p(a)}{p(a)} \rceil = n$. Therefore $r + h * p(a) = r + n * p(a) = r + c(a) - r = c(a) < c(a) + p(a) - g + (d(a) \mod g)$.

If $m > 0$ then $h = \lceil \frac{n*p(a)+m}{p(a)} \rceil = n + 1$ and $r + h * p(a) = r + (n+1) * p(a) = r + c(a) - r - m + p(a) = c(a) + p(a) - m$. Since $c(a) - r = n * p(a) + m$, $m = c(a) - n * p(a) - r$. Since $r = d(a) + t * g$ $m = c(a) - n * p(a) - t * g - d(a) = u * g - d(a) = v * g - (d \mod g)$, where t, u and v are integer values. Since $m > 0$ and $v * g - (d \mod g) > 0$, $v \geq 1$. Therefore $m \geq g - (d \mod g)$. Hence $c(a) + p(a) - m \leq c(a) + p(a) - g + (d \mod g)$.

□

THEOREM 1. *Source-dlcSAS generates a buffer optimal schedule for a chain structured graph.*

PROOF PROOF BY INDUCTION. Assume that a graph has $\{v_1, v_2, ..., v_{n+1}\}$ vertices and $\{a_1, a_2, ..., a_n\}$ arcs. as shown in Figure 7.

(i) If $n = 1$ then arc a_1 requires minimum buffer proved by Lemma 1.

(ii) Assume that if $n = k$ then the algorithm generates buffer optimal schedule for Figure 7 (b). The schedule is $h_{a_k}(h_{a_{k-1}}((...)v_{k-1}))v_k$. Let the schedule S.

When $n = k + 1$, the schedule of $(h_{a_{k+1}}S)v_{k+2}$ does not change S. Therefore the sizes of buffers on $a_1, ..., a_k$ do not change. Since the invocation of S is equal to that of v_{k+1}, optimal buffer for arc a_{k+1} is allocated by Lemma 1. Hence if $n = k+1$, the schedule requires minimum buffer memory.

□

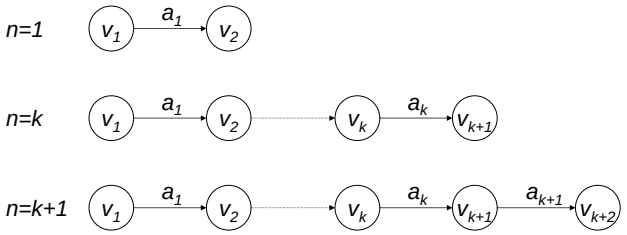


Figure 7: Proof of Theorem 1

Similarly, we can prove that sink-dlcSAS algorithm generates buffer optimal schedule. Moreover, we can mix source-dlcSAS and sink-dlcSAS to generate optimal schedule unless both source and sink loop count are applied for one node as shown in Figure 5.

5.4 Dynamic Loop Count for Arbitrary SDF Graphs

Now, we extend the algorithm to the general graph topology. Since buffer optimal schedule for general SDF graphs is an NP problem [13], we propose a heuristic algorithm. The key idea is to make a sequential list of nodes by adding virtual arcs between two unconnected nodes and apply the source-dlcSAS or the sink-dlcSAS algorithm. In order to make a virtual arc between two unconnected nodes, we use the number of invocations of node v $inv(v)$ to compute the number of producing and consuming sample rates $p(a)$ and $c(a)$. We can prove that ${}_{inv(sink(a))}h_{inv(src(a))}$ is equal to ${}_{p(a)}h_{c(a)}$ by Theorem 2.

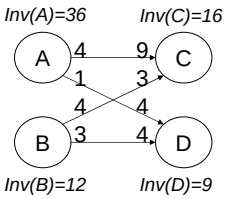
LEMMA 2. ${}_ph_c^d = {}_{n*p}\, h_{n*c}^{n*d}$

PROOF. ${}_ph_c^d = \lceil \frac{c-r}{p} \rceil$ (initially r=d) $= \lceil \frac{n*c-n*r}{n*p} \rceil$ (initially n*r = n*d) $= {}_{n*p}h_{n*c}^{n*d}$ □

THEOREM 2. ${}_{inv(sink(a))}h_{inv(src(a))}$ *is equivalent to* ${}_{p(a)}h_{c(a)}$.

PROOF. Since $inv(src(a)) * p(a) = inv(sink(a)) * c(a)$ by SDF equation, $inv(src(a)) = n * c(a)$ and $inv(sink(a)) = n * p(a)$. ${}_{inv(sink(a))}h_{inv(src(a))} = {}_{n*p(a)}h_{n*c(a)} = {}_{p(a)}h_{c(a)}$ by Lemma 2 □

Theorem 2 says that we may connect two unconnected nodes with a virtual arc of which $p(a)$ and $c(a)$ are computed by invocation numbers.



A: A
B:3AB
C: ${}_4h_3$(3AB)C
D: ${}_9h_{16}({}_4h_3$(3AB)C)D

(b)

```
main()
{
int i,j,AC[21],AD[8],BC[6],BD[8];
int hb=0;rb=0,rc=0,hc=0;
for(;;) {
 for(n=0;n<16;n++) {
  if(rc>=7){hc=1;rc-=7;}
  else{hc=2;rc+=2;}
  for(i=0;i<hc;i++) {
   if(hb==0) {
    for(j=0;j<3;j++)
     {/* A's code */}
    /* B's code */
    if(rb>=2){hb=1;rb-=2;}
    else{hb=0;rb+=1;}
   } else hb--;
   /* C's code */
  }
  /* D's code */
}}}
```

(c)

Figure 8: (a) A general SDF graph, (b) schedule the graph and (c) generated code by dlcSAS

Figure 8 presents an example in which invocation numbers of node A, B, C and D are 36, 12, 16 and 9 respectively. Assume that node A is scheduled before node B, and node C before node D. When scheduling node B after node A, we connect node A and node B with a virtual arc of which $p(a)$ and $c(a)$ are 12 and 36 respectively. Therefore h becomes ${}_{12}h_{36}$ (=3). The intermediate schedule is $(3A)B$. When node C is scheduled, h is ${}_{16}h_{12}(= {}_4h_3)$ since invocations of node B and node C are 12 and 16 respectively. The schedule becomes $({}_4h_3((3A)B))C$. Finally when node D is scheduled, h is ${}_9h_{16}$ since invocations of node C and node D are 16 and 9. Consequently the final schedule becomes $({}_9h_{16}({}_4h_3((3A)B)C))D$ which requires buffers of size 43, while the previous SAS algorithm produces 3(4B3(4AD))16C which requires buffers of size 208.

Now we summarize a dynamic loop count scheduling algorithm for general graphs in Figure 9.

The proposed algorithm first finds a runnable node that has no input arc or enough samples for all of its input arcs. In order to avoid allocating a new loop count variable, a node of which invocation number is equal to that of recently scheduled node and delay is zero is preferred. If the selected node does not need a new loop count then it is inserted to the current loop; otherwise we allocate a new loop count for the node.

When we select a node requiring a new loop count, we prefer a node with the maximum number of invocations. Consider an example of Figure 10(a). If we first select a node with the minimum number of invocations then we can get a schedule in Figure 10(b). In that case, the buffer size of arc BC is 3 since the loop count for node B is {3,0,0}. On the other hand, if node A is scheduled first then the buffer size of arc BC is 1.

The handling of arcs with delay samples in a general graph is more complex than in a chain structured graph. If a scheduled node has a unscheduled child node then delays should be regarded as 0. Consider Figure 11. If a schedule becomes Figure 11 (b) then the second invocation of node

```
main()
{
  int cInv=0;
  Schedule S;
  Node v, lastv;

  do {
    lastv = v;
    v=findRunnableNode(G,cInv,S);
    if(v==null) break;
    fire(v,G);
    if(S.size()==0) {
      S.append(v);
      cInv=inv(v);
    } else if(cInv==inv(v) && delay(v,lastv,S)==0) {
      S.append(v);
    } else {
      S.encloseLoop(n*inv(v),n*cInv, n*delay(v,lastv,S));
      S.append(v);
      cInv = inv(v);
    }
  } while(true);
}
```

$delay(v, lastv, S) = 0$ if $\exists e$ such that $src(e) \in S$,$sink(e) \notin S$ and ($src(e) \neq lastv$ or $sink(e) \neq v$);
$delay(v, lastv, S) = inv(v)* \min(\frac{d(a)}{p(a)})$ while $src(a)$=$lastv$ and $sink(a)$=v, otherwise.
In addition, n is an integer making $n * delay(v, lastv, S)$ integer.

Figure 9: a dynamic loop count scheduling algorithm for general graphs

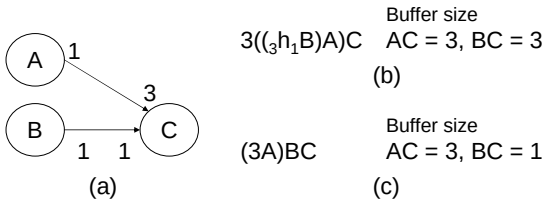


Figure 10: (a) A general SDF graph, (b) minimum invocation node first, and (c) maximum invocation node first

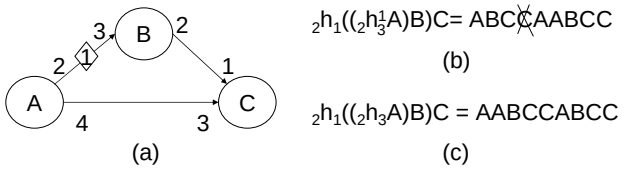
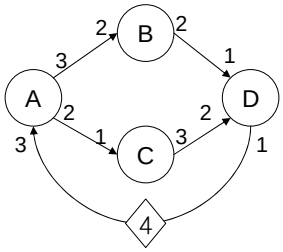


Figure 11: (a) A general SDF graph, (b) wrong schedule, and (c) right schedule

C is impossible since node A should be executed twice in order to execute node C two times. In other words, since ${}_2h_1({}_2h_3^1A) < {}_4h_3A$, node A is not guaranteed to produce enough samples for node C in Figure 11 (b) schedule. Therefore delay for dynamic loop count is regarded as zero when any scheduled node has a unscheduled child node. Moreover, when there are multiple arcs between $lastv$ and v in Figure 9, the minimum delay can be computed by min $\frac{d(a)}{p(a)}$ where $src(a) = lastv$ and $sink(a) = v$ since ${}_{p(a)}h_{c(a)}^{d(a)} = {}_1h_{\frac{c(a)}{p(a)}}^{\frac{d(a)}{p(a)}}$ by Lemma 2 and $\frac{d(a)}{p(a)}$ indicates normalized delay value. We know that ${}_1h_{\frac{inv(src(a))}{inv(sink(a))}}^{d} = {}_{inv(sink(a))*n}h_{inv(src(a))*n}^{d*inv(sink(a))*n}$ where $d = \min(\frac{d(a)}{p(a)})$.



$6({}_2h_1({}_3h_4({}_2h_1A)C)B)D)$

(b)

```
main() {
int i,n,hA=0,hB=0,hC;
int rC=0,DA[4],AB[4],AC[2]
,BD[2],CD[5];
for(;;) {
  for(n=0;n<6;n++) {
    if(hB==0) {
      hB=1;
      if(rC>=1) {hC=1; rC-=1; }
      else {hC=2; rC+=2; }
      for(i=0;i<hC;i++) {
        if(hA==0) {
          hA=1;
          /* A */
        } else hA--;
        /* C */
      }
      /* B */
    } else hB--;
    /* D */
}}}
```

(c)

Figure 12: (a) cyclic SDF graph (b) dlcSAS and (c) generated code by dlcSAS

Figure 12(a) indicates a cyclic graph. Since there are 4 initial samples on arc DA, node A is schedulable. After node A is scheduled, node B and node C are available. We schedule node C first since we prefer a node with a large invocation number. In this example, $inv(C)$=4 and $inv(B)$=3. Since $inv(A)$=2 and $inv(C)$=4, the intermediate schedule becomes $({}_2h_1A)C$. Only node B is available since node D waits for samples on arc BD. The schedule becomes $({}_3h_4(({}_2h_1A)C))B$ since $inv(B)$= 3 and $inv(C)$=4. Finally, after scheduling node D of which invocation number is 6, we can build the schedule of $6(({}_2h_1({}_3h_4({}_2h_1(A)C))B)D)$ as shown in Figure 12(b). Since the schedule is equivalent to ACCBDDACBDDCBDD, buffer sizes on arc DA, AB, AC, BD and CD are 4, 2, 4, 2 and 6 respectively. The buffer size requirement is not optimal since buffer size on arc CD is 4 in the optimal schedule of ABCDCDABDCDBCDD. Note that if node B precedes node C then a schedule is $6({}_3h_2({}_4h_3({}_3h_2(A)B)C)D)$ and is not valid since it needs 5 initial samples on arc DA.

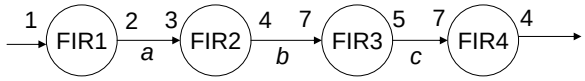
Even though the dlcSAS is not optimal for general SDF graphs, it still requires less buffers than the previous single appearance schedule. In this example, the previous single appearance schedule cannot generate any valid schedule

since it requires 6 initial samples on arc DA to execute node A twice. If there are 6 samples on arc DA, the buffer requirements on arc DA, AB, AC, BD and CD become 6, 6, 2, 2 and 12 from a schedule of $2(A(2C))3(B(2D))$ computed by APGAN.

6. EXPERIMENTS

We have implemented the proposed algorithm into our high level system design framework [2] and experimented two real life examples to demonstrate effectiveness of our approach. In these experiments, we used the arm compiler and armulator to measure memory size and cycles on ARM920T processor. The first example is compact dist to digital audio tape converter(CD2DAT) and the second example is 4 channel non-uniform filter bank.

Figure 13 illustrates a CD2DAT application in an SDF 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 live samples while each node has additional buffers if it need to refer previous samples. The invocations of FIR1, FIR2, FIR3 and FIR4 are 147, 98, 56 and 40 respectively. In this application, the previous SAS algorithm required 6, 56, and 280 for each arc totaling 342 while the proposed dlcSAS algorithm needs 4,10,11 size buffers, which is optimal.



(a)

SAS : 7(7(3(FIR1)2(FIR2))8(FIR3))40(FIR4)
dlcSAS:40($_5h_7$ ($_4h_7$ ($_2h_3$(FIR1)(FIR2))(FIR3))(FIR4))

(b)

```
main()
{
 int i,i1,i2,i3,r1=0,r2=0,r3=0, n1,n2,n3, a[4],b[10],c[11];
 for(;;) {
    for(i=0;i<40;i++) {
       if(r3>=2) {n3=1; r3-=2;} else {n3=2; r3+=3;}
       for(i3=0;i3<n3; i3++) {
          if(r2>=3) { n2=1; r2-=3;} else {n2=2; r2+=1;}
          for(i2=0; i2<n2; i2++) {
             if(r1>=1) {n1=1; r1-=1;} else {n1=2; r1+=1;}
             for(i1=0; i1<n1; i1++) {
                /* FIR1's code */    }
             /* FIR2's code */    }
          /* FIR3's code */  }
       /* FIR4's code */
}}}
```

(c)

Figure 13: (a) A CD2DAT algorithm, (b) schedules, and (c) generated code by dlcSAS

Table 1 summarizes the comparison results between the previous SAS approach [13] and proposed dlcSAS approach on ARM920T processor. Our schedule generates a code saving 40% data memory with 0.03% and 1.54% execution time and code size overheads respectively. These overheads are quite small compared with 0.75% and 10.85% execution time and code size overhead respectively on TMS320C67x in the function call invocation approach [9].

Table 1: Comparison for CD2DAT example

	previous SAS	dlcSAS	ratio(%)
code memory	7516 bytes	7632 bytes	1.54
data memory	6724 bytes	4032 bytes	-40.04
total memory	14240 bytes	11664 bytes	-18.09
cycles	54981K cycles	54998K cycles	0.03

Table 2: Comparison for non-uniform filter bank example

	previous SAS	dlcSAS	ratio(%)
code memory	13128 bytes	13540 bytes	3.14
data memory	15720 bytes	9664 bytes	-38.52
total memory	28848 bytes	23204 bytes	-19.56
cycles	71060K cycles	71363K cycles	0.43

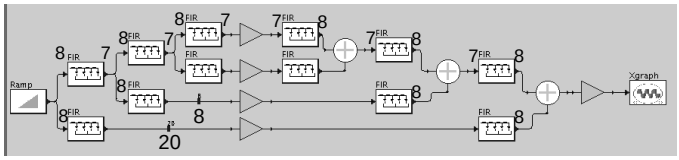


Figure 14: SDF graph for a non-uniform filterbank. The highpass channel retains 1/8 of the spectrum and the lowpass channel retains 7/8 of the spectrum

Figure 14 represents the SDF graph of a 4-channel non-uniform filterbank. The sample rates are shown on each arch whenever they are different from unity. In the 4-channel non-uniform filterbank, the lowpass filters retain 7/8 of the spectrum while the highpass filters retain 1/8. We can also save more than 20% total memory with less than 1% performance overhead in this example.

7. CONCLUSION

In this paper, we presented a new single appearance scheduling algorithm to minimize data memory and code memory jointly for synchronous dataflow graphs. Our algorithm is different from previous algorithms in terms of determining loop counts at run time even though the SDF graphs can be scheduled at compile time. Therefore while it introduces performance overhead to compute loop counts(which is much lower than function call approaches), it reduces buffer memory requirement close to buffer lower bounds of non single appearance schedule. Especially when the given graph is a chained structure, the proposed algorithm guarantees memory optimal schedule. In case of cd2data and non uniform filter bank applications, we can reduce more than 20% of total memory size with less than 1% performance overhead compared with the previous single appearance schedules.

8. ACKNOWLEDGMENTS

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.

9. REFERENCES

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

[2] *http://peace.snu.ac.kr/research/peace*.

[3] M. Ade, R. Lauwereins, and J. A. Peperstraete. Buffer memory requirements in dsp applications. In *IEEE Wkshp. on Rapid System Prototyping*, June 1994.

[4] M. Ade, R. Lauwereins, and J. A. Peperstraete. Data memory minimization for synchronous data flow graphs emulated on dsp-fpga targets. In *DAC*, June 1997.

[5] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publisher, Norwell MA, 1996.

[6] S. S. Bhattachayya, P. K. Murthy, and E. A. Lee. Apgan and rpmc: Complementary heuristics for translating dsp block diagrams into efficient software implementations. In *Journal of Design Automation for Embedded Systems*, volume 2, pages 33–60, January 1997.

[7] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschimitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. In *Int. Journal of Computer Simulation, special issue on Simulation Software Development*, volume 4, pages 155–182, April 1994.

[8] M. Karczmarek, W. Thies, and S. Amarasinghe. Phased scheduling of stream programs. In *LCTES'03*, June 2003.

[9] M. Ko, P. K. Murthy, and S. S. Bhattacharyya. Compact procedural implementation in DSP software synthesis through recursive graph decomposition. In *Proceedings of the International Workshop on Software and Compilers for Embedded Processors*, pages 47–61, Amsterdam, The Netherlands, September 2004.

[10] R. Lauwereins, M. Engels, J. A. Peperstraete, E. Steegmans, and J. V. Ginderdeuren. Grape: A case tool for digital signal parallel processing. In *IEEE ASSP Magazine*, volume 7, pages 32–43, April 1990.

[11] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous dataflow programs for digital signal processing. In *IEEE Transaction on Computer*, volume C-36, pages 24–35, January 1987.

[12] P. K. Murthy and S. S. Bhattacharyya. Shared buffer implementations of signal processing systems using lifetime analysis techniques. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 20, pages 177–198, February 2001.

[13] P. K. Murthy, S. S. Bhattachayya, and E. A. Lee. Joint minimization of code and data for synchronous dataflow programs. In *Journal of Formal Methods in Systems Design*, volume 11, pages 41–70, July 1997.

[14] H. Oh and S. Ha. Memory-optimized software synthesis from dataflow program graphs with large size data samples. In *EURASIP Journal on Applied Signal Processing*, volume 2003, pages 514–529, May 2003.

[15] S. Ritz, M. Willems, and H. Meyr. Optimum vectorization of scalable synchronous dataflow graphs. In *Proceedings of the International Conference on Application-Specific Array Processors*, October 1993.

[16] S. Ritz, M. Willems, and H. Meyr. Scheduling for optimum data memory compaction in block diagram oriented software synthesis. In *Proceedings of the ICASSP 95*, May 1995.

[17] W. Sung and S. Ha. Memory efficient software synthesis using mixed coding style from dataflow graph. In *IEEE Transaction on VLSI Systems*, volume 8, pages 522–526, October 2000.

[18] E. Zitzler, J. Teich, and S. S. Bhattacharyya. Evolutionary algorithm based exploration of software schedules for digital signal processors. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1762–1770, Orlando, Florida, USA, 13-17 1999. Morgan Kaufmann.