Memory Optimal Single Appearance Schedule with Dynamic Loop Count for Synchronous Dataflow Graphs

Hyunok Oh, Nikil Dutt
Center for Embedded Computer Systems
University of California, Irvine, CA
{hoh,dutt}@ics.uci.edu

Soonhoi Ha
School of EECS
Seoul National University, Seoul, 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 simultaneously. While a single appearance schedule promises only one appearance of each node definition in the generated code, it requires significant amount of data memory overhead compared with a buffer optimal schedule allowing multiple appearance. 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 schedule produces a single appearance schedule code with minimum data memory requirement. We prove that every buffer optimal schedule can be transformed to our single appearance schedule which requires optimal buffer size for arbitrary synchronous dataflow graphs. The only penalty for the proposed technique is slight performance overhead of computing loop counts dynamically. In order to minimize the overhead we propose optimization techniques. 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.

I. 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 [6], and Ptolemy [4]. It is also the main concern of this paper.

In a hierarchical dataflow program graph, a node, called a actor 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) [7] 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.

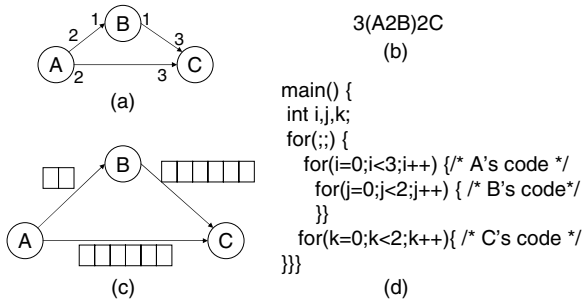


Fig. 1. (a) SDF graph example, (b) a scheduling result, (c) a code template, and (d) buffer allocation

To generate a code from the given SDF graph, the order of node executions is determined at compile time by static *scheduling* of the graph. Since a dataflow graph specifies only partial orders between nodes, 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

0-7803-9451-8/06/$20.00

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 II defines some notations and section III reviews the related works. In section IV, we introduce motivational examples. The proposed technique is explained in section V. We will show experimental results in section VI and make a conclusion in section VII.

II. 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

$inputArc(v)$: a set of incoming arcs to node v.

$outputArc(v)$: a set of outgoing arcs from node v.

For arc AB in Figure 1, $src(AB) = A$, $sink(AB)=B$, $p(AB)=2$, $c(AB)=1$, $d(AB)=0$, $outputArc(A) = \{AB, AC\}$, $inputArc(B) = \{AB\}$,$outputArc(B) = \{BC\}$,and $inputArc(C) = \{AC, BC\}$.

III. 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. [2] 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 [9] which has shown that the problem of constructing a schedule that minimizes the buffer requirement is NP-complete.

Ritz et al. [12] 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 [9]. 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 [13, 5]. These approaches observe the tradeoff 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 [8, 11] 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.

Dynamic loop count schedule for a chained structure graph has been developed, which requires optimal buffer size [10]. However the schedule is not optimal for general graphs with delay samples and feedback arcs. In this paper, we extend the previous schedule to memory optimal schedule for arbitrary SDF graphs.

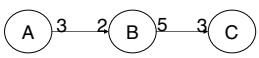
IV. Motivation

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 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. 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 pre-

serving the minimum buffer memory as the buffer optimal non SAS.



(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],iC=0;
 int lB[2]={1,2},lC[3]={1,2,2};
 for(;;) {
   for(n=0;n<2;n++) {
     /* A's code */
     for(i=0;i<lB[n];i++) {
      /* B's code */
      for(j=0;j<lC[iC];j++)
      { /* C's code */}
      iC=(iC+1)%3; }
}}}
```

(c)

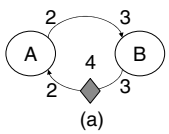
Fig. 2. (a) An SDF graph (b) schedule results and (c) generated code by dlcSAS

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 non SAS. In order to transform non SAS to dlcSAS, we first choose the appearance order of each node by applying topological sort. And then, we determine the loop count of each node by comparing the non SAS with the appearance order.

For instance, assume that ABACABBD schedule is given. We choose the appearance order as "ABCD". By comparing AB with ABCD, we make a schedule of {1}A{1}B{0}C{0}D. By comparison of AC with ABCD, we build a schedule of {1,1}A{1,0}B{0,1}C{0,0}D. Finally, the schedule becomes {1,1,1}A{1,0,2}B{0,1,0}C{0,0,1}D by comparing ABBD with ABCD.

Even though we can build dlcSAS equivalent to any schedule, we are interested in a code with simple expression of loop count computation to minimize code memory and performance overhead. In the following section, we will discuss how to compute the loop count with simple computation.



(a)

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

(b)

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

(c)

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

V. Dynamic Loop Count Single Appearance Scheduling Algorithm

A. Dynamic Loop Count for a Chained Structure Graph

In this section, we briefly explain dlcSAS for chained structure in the previous paper [10].

When we change the loop count value of a source node, the source node should be executed multiple times to produce the sufficient number of samples for the sink node. Let r be the accumulated number of samples on the arc. After the source node is executed h times, $h * p(a) + r$ samples are available. Since $h * p(a) + r \geq c(a)$ in order to execute the sink node, h becomes $\lceil \frac{c(a)-r}{p(a)} \rceil$. After executing both nodes, the accumulated number r is updated as r+$h * p(a) - c(a)$.

When we change the loop count value of a sink node, the sink node can be executed until the accumulated samples are exhausted. Let k be the loop count of the sink node. Since there are $r + p(a)$ samples after execution of the source node and $r + p(a) - k * c(a) \geq 0$, k becomes $\lfloor \frac{r+p(a)}{c(a)} \rfloor$. After executing both nodes, there are $r + p(a) - k * c(a)$ samples on the arc. Note that we use ${}_{p(a)}h_{c(a)}$ and ${}_{p(a)}k_{c(a)}$ to represent loop counts.

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* $r = r + p(a) - k * c(a)$.

B. Dynamic Loop Count for General Graphs

Now we explain the main dlcSAS algorithm for general graphs. It is complex to minimize data buffer memory size for a arbitrary graph. While for an acyclic graph without delay samples an algorithm of which time complexity is $O(e^3)$ have been developed [2] where e is the number of arcs, the buffer minimum scheduling becomes NP [9] for a graph with delay samples. Therefore it is practically impossible to build a schedule with simple expression of loop count computation without knowing buffer size.

Fortunately, when we know buffer size on each arc at compile time we can compute loop count for each node at run time with slight performance overhead. Therefore we assume that the buffer size for each arc is already computed at compile time by using existing heuristics in this paper.

Since a node consumes samples on input arcs and produces samples onto output arcs, it can be executed while there are sufficient samples on all input arcs and the number of samples on every output arc does not exceed the given buffer size.

Since a node can be executed while there are enough samples on all of its input arcs, the maximum loop count k for the node becomes minimum value among the number of live samples ($r(e_i)$) over the consuming rate ($c(e_i)$) on arc e_i that is an input arc. Since it consumes $k * c(e_i)$ samples that should be

no greater than the number of samples $r(e_i)$ on arc e_i when the node is executed k times, $k * c(e_i) \leq r(e_i)$ and $k \leq \frac{r(e_i)}{c(e_i)}$. Moreover the node can produce samples while the number of samples does not exceed buffer size on every arc. When the node is executed h times, the number of produced samples is $h * p(e_j)$. Since there are $r(e_j)$ samples, the total number of samples is $h * p(e_j) + r(e_j)$ that should be no greater than buffer size $bs(e_j)$ on arc e_j. So $h * p(e_j) + r(e_j) \leq bs(e_j)$ and $h \leq \frac{bs(e_j) - r(e_j)}{p(e_j)}$.

A loop count computation for a node A is summarized as following:

Equation 2.

$$LoopCount = \min(k, h)$$

$$k = \min_{e_i \in inputArc(A)} \lfloor \frac{r(e_i)}{c(e_i)} \rfloor$$

$$h = \min_{e_j \in outputArc(A)} \lfloor \frac{bs(e_j) - r(e_j)}{p(e_j)} \rfloor$$

$$For\ e_i \in inputArc(A), e_i - = LoopCount * c(e_i)$$

$$For\ e_j \in iutputArc(A), e_j + = LoopCount * p(e_j)$$

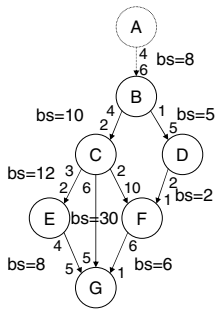
where $r(e_i)$ indicates the number of remained samples on arc e_i.

In Equation 2, k denotes loop count by considering input arcs and h considering output arcs. Since the final loop count is constrained by the number of samples on the input arcs and the remained buffer size on the output arcs, the loop count becomes minimum number between k and h. After determining the loop count, we update the number of samples on each arc connected with the node. By using Equation 2, we can build memory optimal dlcSAS for arbitrary SDF graphs.

Theorem 1. *Every buffer optimal schedule for synchronous dataflow graphs can be transformed to an equivalent dynamic loop count single appearance schedule that requires same buffer size.*

Since we consider all arcs to compute loop count with preserving the number of samples, the run time overhead of the proposed scheduling is proportional to the number of arcs in the given graph.

Consider Figure 4(a) in [2], in which *bs* on each arc indicates optimal buffer size. Since the subgraph of node A and B is chained structure, the algorithm for chained structure is applied. Therefore we build $(_4h_6\text{A})\text{B}$ schedule. For the remained nodes, we need to apply the scheduling algorithm for general graphs. First we determine lexical appearance order by applying topological sort. In this example, the order becomes (AB) C D E F G. Note that we use topological ordering to minimize performance overhead by minimizing zero loop count even though any ordering is applicable. Since the loop count of node B is dependent on arc BC and BD, the loop count denoted l_B is the minimum between (10-rBC)/4 and (5-rBD) where rBC and rBD indicate the number of samples on arc BC and BD respectively. Similarly, we can compute loop counts



(a)

$l_B((_4h_6\text{A})\text{B})\ l_C(\text{C})\ l_D(\text{D})\ l_E(\text{E})\ l_F(\text{F})\ l_G(\text{G})$

l_B=min(h_{BC},h_{BD}),l_C=min(k_{BC},h_{CE},h_{CF},h_{CG}),
l_D=min(k_{BD},h_{DF}),l_E=min(k_{CE},h_{EG}),
l_F=min(k_{CF},k_{DF},h_{FG}), l_G=min(k_{CG},k_{EG},k_{FG})

(b)

```
main() {
int I,j,IA,IB,IC,ID,IE,IF,IG;
int rA=0;rBC=0,rBD=0,rCE=0;
Int rCF=0,rCG=0,rDF=0,rEG=0,rFG=0;
for(;;) {
 IB = min((10-rBC)/4,(5-rBD));
 rBC+=4*IB; rBD+=IB;
 for(i=0;i<IB;i++) {
  if(rA <2) {IA=2;rA+=2;} else {IA=1;rA-=2;}
  for(j=0;j<IA;j++) /* A */
  /* B */ }
 IC = min(BC/2,(12-rCE)/3,(10-rCF)/2,(30-rCG)/6);
 rBC-=2*IC;rCE+=3*IC;rCF+=2*IC;rCG+=6*IC;
 for(i=0;i<IC;i++) /* C */
 ID = min(rBD/5,(2-rDF)/2);
 rBD-=5*ID; rDF+=2*ID;
 for(i=0;i<ID;i++)  /* D */ }
 IE = min(rCE/2,(8-rEG)/4);
 rCE-=2*IE;rEG+=4*IE;
 for(i=0;i<IE;i++) /* E */
 IF=min(rCF/10,rDF,(6-rFG)/6);
 rCF-=10*IF;rDF-=IF;rFG+=6*IF;
 for(i=0;i<IF;i++) /* F */
 IG=min(rCG/5,rEG/8,rFG);
 for(i=0;i=IG;i++) /* G */
}}
```

(c)

Fig. 4. (a) A general SDF graph, (b) schedule the graph and (c) generated code by dlcSAS

for the other nodes by applying Equation 2 as shown in Figure 4(b). Finally we can generate a code as shown in Figure 4(c).

C. Optimization of Schedule for General Graphs

Since the loop count of a node is constrainted by the accumulated samples and buffer size on each connected arc, $2 * e$ computations are required where e denotes the number of arcs in the graph. Some constraints on arcs, however, can be eliminated when the constraints have been never used. Hence optimization techniques eliminate unnecessary constraints.

First we examine the loop count computation dependency by running the unoptimized schedule for an iteration period. If the number of samples on an arc is not used for computation of a loop count then the expression referring to the arc can be eliminated. Furthermore, when no node refers to an arc, variables on the arc are removed.

If a loop count is computed by an expression on an arc, dlcSAS for chained structure graphs is applied. If the loop count of a node is dependent on its output arc then Equation 1(i) is applied. It is, however, not applicable when the loop count is not zero even if the loop count of the sink node is zero. For instance, the schedule of ABCAC = $\{1,1\}$A $\{1,0\}$B $\{1,1\}$C cannot be represented by $l_B((h\ A)B)l_C C$ since $(h\ A)B$ cannot express $\{1\}$A $\{0\}$B schedule. Similarly, Equation 1(ii) is applicable if the loop count of the sink node is only dependent on its input arc and its loop count becomes zero whenever its source node loop count is zero.

In addition, if the loop count only has 0 or 1 then more compact code can be generated.

We can summarize the optimization techniques as follows:

Algorithm

1: Run unoptimized dlcSAS for an iteration period.

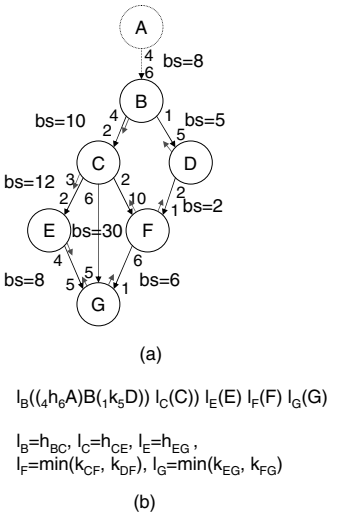
2: Eliminate a loop count computation if the computation is not used to compute minimum loop count value.

3: Eliminate the updating code of the number of samples which is not referred to in a loop count computation.

4: For arc a, if the loop count $l_{src(a)}$ of source node $src(a)$ is only dependent on arc a and $l_{src(a)}$=0 whenever $l_{sink(a)}$=0 then Equation 1(i) is applied. Similarly if $l_{sink(a)}$ is only dependent on arc a and $l_{sink(a)}$=0 whenever $l_{src(a)}$=0 then Equation 1(ii) is used.

5: If loop count l_A of node A has only 0 or 1 then "if-statement" code is generated instead of "for-loop" as following:

if($\bigwedge_{e_i \in outputArc(A)} (r(e_i) \leq bs(e_i) - p(e_i))$
$\bigwedge_{e_j \in inputArc(A)} (r(e_j) \geq c(e_j)))$
{
For all $e_i \in outputArc(A), r(e_i) += p(e_i)$;
For all $e_j \in inputArc(A), r(e_j) -= c(p_j)$;
/* A's code */
}




```
main() {
int i1,i2,IA,IB,IC,ID,IE,IF,IG;
int rA=0;rD=4,rBC=0,rCE=0;
Int rCF=0,rDF=0,rEG=0,rFG=0;
for(;;) {
  IB = (10-rBC)/4;
  rBC+=4*IB;
  for(i1=0;i1<IB;i1++) {
    if(rA <2) {IA=2;rA+=2;} else {IA=1;rA-=2;}
    for(i2=0;i2<IA;i2++) /* A */
    /* B */
    if(rD==0) {rD=4; rDF+=2; /* D*/ } else rD--;
  }
  IC = (12-rCE)/3;
  rBC-=2*IC;rCE+=3*IC;rCF+=2*IC;
  for(i1=0;i1<IC;i1++) /* C */
  IE = (8-rEG)/4;
  rCE-=2*IE;rEG+=4*IE;
  for(i1=0;i1<IE;i1++) /* E */
  if(rCF>=10 && rDF>=1)
  { rCF -= 10; rDF--; rFG+=6; /* F */ }
  if(rEG>=5 && rFG >=1)
  { rEG-=5;rFG-=1; /* G */ }
}}
```

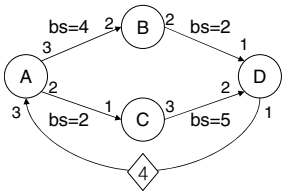
(c)

Fig. 5. (a) Examine whether each arc contributes loop count computation or not. Short arrows indicate the loop count of the node is dependent on the arc. (b) Optimized schedule and (c) optimized code by dlcSAS

Now, we apply the optimization techniques to Figure 4 (a) graph. During simulating the code of Figure 4 (c), we inspect which arcs each loop count is dependent on. In Figure 5 (a), short arrows represent the dependency of loop count on arcs. The loop count of node B is $(10-rBC)/4$ ignoring $(5-rBD)$ since it is dependent on arc BC only. Node C just requires rCE value to compute its loop count and l_C=h_{CE}=$(12-rCE)/3$ although it has three arcs BC, CE and EF. Similarly, we know that loop count of node D is dependent on rBD, node E is on rEG, node F on rCF and rDF, and node G on rEG and rFG. Since no node refers to arc CG, rCG is not necessary to be maintained.

Figure 5(b) represents optimized schedule. By the optimization, we can reduces loop count computations from 16 expressions to 7 expressions. Furthermore, we can eliminate 8 conditional expressions to find minimum values from 10 conditions in the unoptimized schedule.

Since the loop counts of node F and node G have only two values of 0 and 1, the generated code is more compact as shown in Figure 5 (c).



$l_B((_3h_2A)B)\ l_C(C)\ l_D(D)$

l_B=h_{BD},l_C=h_{CD}, l_D=min(k_{BD},h_{CD})

(b)

```
main() {
int i,lB,lC,lD;
int rA=0;rBD=0,rCD=0;
for(;;) {
 if(rBD<=0) {
   if(rA<=1) { rA+=1; /* A */ }
   else { rA-=2; }
   rBD += 2;
   /* B */
 }
 if(rCD<=2) {
   rCD+=3;
   /* C */
 }
 lD = min(rBD,rCD/2);
 rBD-=lD;rCD-=2*lD;
 for(i=0;i<lD;i++) /* D */
}}
```

(c)

Fig. 6. (a) cyclic SDF graph (b) Optimized schedule and (c) optimized code by dlcSAS

Figure 6(a) indicates a graph with a cycle. First, it acquires buffer size on each arc by using existent heuristics. And then we determine the lexical appearance order by applying topological sort. In this example, assume that the order is ABCD. By running a graph with $(l_A A)(l_B Br)(l_C C)(l_D D)$ schedule, we examine the dependency of loop count. This example shows that node A can be clustered into node B since the loop count of node A is only dependent on arc AB and the loop count of node A is always 0 when that of node B is 0. Figure 6(b) represents the optimized schedule. The loop counts of node B and C rely on arc BD and CD respectively and the loop count of node D on both arc BD and CD. Figure 6(c) shows a generated code with minimal performance overhead for computation of loop counts.

VI. Experiments

We have experimented several examples to demonstrate effectiveness of our approach. Table I represents minimum buffer size for various examples between previous SAS(APGAN) and proposed dlcSAS. The last column indicates the buffer size reduction by dlcSAS compared with the previous SAS, which is computed by (previousSAS-dlcSAS)/previousSAS. For Figure 6 that contains feedback cycle, the previous SAS is not applicable since there are not enough delay samples on arc DA. Note that 6 delay samples are required on arc DA for the previous SAS to produce a schedule. Since sample rate is stable in the modem application [3], both the previous SAS and dlc SAS require same size buffer.

In order to measure memory size and performance overhead on real platform, we used the arm compiler and armulator for ARM920T processor.

Figure 7 represents the SDF graph of a 4-channel non-uniform filterbank. The sample rates are shown on each arch

TABLE I
COMPARISON OF BUFFER SIZE

application	SAS(APGAN)	dlc SAS	reduction(%)
Figure 1	14	9	36
Figure 4	194	81	58
Figure 6	N/A	17	N/A
modem [3]	38	38	0
Figure 5 in [3]	120	28	77

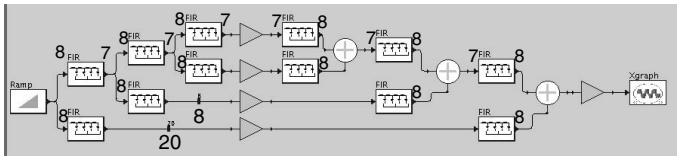


Fig. 7. SDF graph for a non-uniform filterbank. The highpass channel retains 1/8 of the spectrum and the lowpass channel retains 7/8 of the spectrum

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.

VII. 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 to buffer lower bounds of non single appearance schedule for arbitrary graphs. Therefore we can argue that the proposed schedule is memory optimal. For non uniform filter bank application, we can reduce more than 20% of total memory size with less than 1% performance overhead compared with the previous single appearance schedules.

In the future, we will extend the schedule to consider buffer sharing.

TABLE II
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

VIII. 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.

REFERENCES

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

[2] 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.

[3] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Synthesis of embedded software from synchronous dataflow specifications. In *Journal of VLSI Signal Processing*, volume 21, pages 151–166, June 1999.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] H. Oh, N. Dutt, and S. Ha. Single appearance schedlue with dyanmic loop count for sy. In *CASES2005*, volume 2005, pages 514–529, Sept 2005.

[11] 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.

[12] 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.

[13] 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.