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 [8], and Ptolemy [6]. 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) [9] 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.

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)(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 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. [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 [11] 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 [11]. 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, 7]. 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 [10, 2] 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 [3]. 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 ABCABCBC(=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 preserving the minimum buffer memory as the buffer optimal non SAS.

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

For the simple examples discussed above, \textit{dlcSAS} may be regarded as a different representation of non SAS. In order to transform non SAS to \textit{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 and 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 ABBBD with ABCD.

Even though we can build \textit{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.

V. DYNAMIC LOOP COUNT SINGLE APPEARANCE SCHEDULING ALGORITHM

A. Dynamic Loop Count for a Chained Structure Graph

In this section, we briefly explain \textit{dlcSAS} for chained structure in the previous paper [3].
samples is \( h \cdot p(e_j) + r(e_j) \) that should be no greater than buffer size \( bs(e_j) \) on arc \( e_j \). So \( h \cdot p(e_j) + r(e_j) \leq bs(e_j) \) 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.**

\[
\text{LoopCount} = \min(k, h)^i \]

\[
k = \min_{e_i \in \text{input Arc}(A)} \left( \frac{r(e_i)}{c(e_i)} \right) \]

\[
h = \min_{e_j \in \text{output Arc}(A)} \left( \frac{bs(e_j) - r(e_j)}{p(e_j)} \right)
\]

For \( e_i \in \text{input Arc}(A), e_i^- = \text{LoopCount} \cdot c(e_i) \)

For \( e_j \in \text{output Arc}(A), e_j^+ = \text{LoopCount} \cdot p(e_j) \)

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 dlSAS 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 [4], 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 \((ch A)B\) 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 \((10rBC)/4\) and \((5rBD)\) where rBC and rBD indicate the number of samples on arc BC and BD respectively. Similarly, we can compute loop counts 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 constrained by the accumulated samples and buffer size on each connected arc, \( 2 \cdot e \) 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, dlSAS 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 \( l_B((hA)B)C \) since \((hA)B\) 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 dlSAS 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 $l_{src(a)}=0$ 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:

\[
\text{if}(\bigwedge_{e_i\in outputArc(A)} (r(e_i) < bs(e_i) - p(e_i))) \\wedge \bigwedge_{e_j\in inputArc(A)} (r(e_j) \geq c(e_j)))
\]

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 */

Fig. 5(b) represents optimized schedule. By the optimization of loop counts, 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.

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 $(l_A A)\langle B Br\rangle\langle C C\rangle\langle D D\rangle$ 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.

Figures 5 and 6 show the 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, 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 arc 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.
TABLE I
COMPARISON OF BUFFER SIZE

<table>
<thead>
<tr>
<th>application</th>
<th>SAS(APGAN)</th>
<th>dlc SAS</th>
<th>reduction(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>14</td>
<td>9</td>
<td>36</td>
</tr>
<tr>
<td>Figure 4</td>
<td>194</td>
<td>81</td>
<td>58</td>
</tr>
<tr>
<td>Figure 6</td>
<td>N/A</td>
<td>17</td>
<td>N/A</td>
</tr>
<tr>
<td>modem [5]</td>
<td>38</td>
<td>38</td>
<td>0</td>
</tr>
<tr>
<td>Figure 5 in [5]</td>
<td>120</td>
<td>28</td>
<td>77</td>
</tr>
</tbody>
</table>

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 excluding code memory overhead for loop count computation at run time. 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.

REFERENCES


[3] Table II
COMPARISON FOR NON-UNIFORM FILTER BANK EXAMPLE

<table>
<thead>
<tr>
<th></th>
<th>previous SAS</th>
<th>dlc SAS</th>
<th>ratio(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>code memory</td>
<td>13128 bytes</td>
<td>13540 bytes</td>
<td>3.14</td>
</tr>
<tr>
<td>data memory</td>
<td>15720 bytes</td>
<td>9664 bytes</td>
<td>-38.52</td>
</tr>
<tr>
<td>total memory</td>
<td>28848 bytes</td>
<td>23204 bytes</td>
<td>-19.56</td>
</tr>
<tr>
<td>cycles</td>
<td>71060K cycles</td>
<td>71363K cycles</td>
<td>0.43</td>
</tr>
</tbody>
</table>


