Efficient Code Synthesis from Extended Dataflow Graphs for Multimedia Applications

Hyunok Oh Soonhoi Ha
The School of Electrical Engineering and Computer Science
Seoul National University
Seoul 151-742, KOREA
TEL : +82-28807292
{oho,sha}@comp.snu.ac.kr

ABSTRACT

This paper presents efficient automatic code synthesis techniques from dataflow graphs for multimedia applications. Since multimedia applications require large size buffers containing composite type data, we aim to reduce the buffer sizes with fractional rate dataflow extension and buffer sharing technique. In an H.263 encoder experiment, the FRDF extension and buffer sharing technique enable us to reduce the buffer size by 67%. The final buffer size is no more than in a manual reference code.

Keywords

memory optimization, software synthesis, multimedia, dataflow

1. Introduction

As system complexity increases and fast design turn-around time becomes important, automatic code synthesis from dataflow program graphs is a promising high level design methodology for rapid prototyping of complicated multimedia embedded systems. COSSAP[1], GRAPE[2], and Ptolemy[3] are well-known design environments, especially for digital signal processing applications, with automatic code synthesis facility from graphical dataflow programs.

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

Memory efficient code synthesis from static dataflow models has been an active research subject to reduce the gap in terms of memory requirements between the automatically synthesized code and the manually optimized code[5]. In the synthesized code, function codes of atomic blocks are stitched together according to the given execution schedule from the compile-time analysis. And each arc is assigned a buffer memory whose size is at least the maximum amount of samples accumulated on the arc at run time. Many works have been done to minimize the buffer requirements by optimal scheduling[4][5] or by buffer sharing[6][7][8]. While most previous works do not differentiate the composite data type from the primitive data type, we can do much better by taking special account for the composite data type, particularly for multimedia applications.

In this paper, we aim to reduce the buffer size requirements in the automatically synthesized code no more than in the manually optimized code especially for the multimedia applications. At first, we introduce a new dataflow extension called fractional rate dataflow(FRDF) in which fractional number of samples can be produced or consumed.

Composite data types, such as video frame or network packet, are used extensively in recent networked multimedia applications, and become the major consumer of scarce memory resource. However, existent dataflow models have inherent difficulty of efficiently expressing the mixture of a composite data type and its constituents: for example, a video frame and macro blocks. A video frame is regarded as a unit of data sample in integer rate dataflow graphs, and should be broken down into multiple macro blocks, explicitly consuming extra memory space. In the proposed FRDF model, a constituent data type is considered as a fraction of the composite data type. Thus no explicit data type conversion is needed.

Next, we propose a new technique to share buffers among different size of data samples. To our best knowledge, no existent technique deals with this problem if dataflow graphs contain delays. When synthesizing software from an SDF/FRDF graph according to the order of block executions or “schedule”, a buffer array is allocated to each arc to store the data samples between the source and the destination blocks. Buffer sharing can reduce the allocated buffer memory size. Such sharing decision can be made at compile-time through life-time analysis of data samples, which is a well-known compilation technique.

A key difference between the proposed technique and the previous approaches is that we separate the local pointer buffers from

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.

This research is supported by National Research Laboratory Program (number M1-0104-00-0015) and Brain Korea 21 program.
The RIACT at Seoul National University provides research facilities for this study.

global data buffers explicitly in the synthesized code. This separation provides more memory sharing chances especially when the number of local buffer entries becomes more than one. Let us examine Figure 1(a) which illustrates a simplified version of an H.263 encoder algorithm where 'ME' node indicates a motion estimation block, 'Trans' is transform coding which includes DCT and Quantization, and 'InvTrans' consists of inverse transform coding and image reconstruction. Each sample between nodes is a frame of 176x144 byte size which is large enough to ignore local buffer size. The diamond symbol on the arc between 'ME' and 'InvTrans' denotes an initial data sample, which is the previous frame in this example. If we do not separate local buffers from global buffers then we need three frame buffers as shown in Figure 1(b) since buffers *a* and *c* overlap their life-times at 'ME', *a* and *b* at 'Trans', and *b* and *c* at 'InvTrans'; otherwise we can use only two frames. Figure 1(c) shows the allocation of local buffers and global buffers, and the mapping of local buffers to global buffers. The detailed algorithm and code synthesis techniques will be explained in section 4.

In section 2, we discuss the fractional rate dataflow model with an H.263 encoder example. Section 3 explains the overview of the proposed buffer sharing technique. In section 4, we explain how to determine the minimum local buffer sizes and their mappings to the minimum global buffers assuming that all data samples have the same size. The sharing technique is extended to a graph with data samples of the different sizes and delays in section 5. Finally, we present some experimental results in section 6, and make conclusions in section 7.

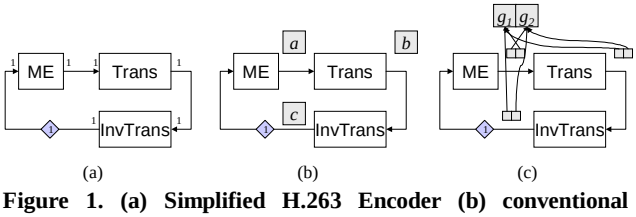

Figure 1. (a) Simplified H.263 Encoder (b) conventional approach (c) separation of global buffer and local buffer

2. Fractional Rate Dataflow

Figure 2 (a) shows an SDF subgraph of an H.263 encoder example. The ME block receives the current and the previous frames to compute the motion vectors and pixel differences, and the D block breaks down the input frame into 99 macro blocks to be encoded in the next EN block. Since the ME block regards a 176x144 video frame as the unit data sample while the EN block a 16x16 macro block, the D block is inserted for explicit data type conversion. This subgraph has the following schedule of block execution: 1(ME,D)99(EN) meaning that after executing the ME and D blocks once, we execute the EN blocks 99 times. Note that the buffer requirements on arcs α and β both have the 176x144 frame size.

Compared with the hand-optimized code with the program graph, we observe that the motion estimation block need not produce the frame-size output at once. Since the ME block is the most time consuming and runs for each macro block, it had better generate output samples at the unit of macro block for shorter latency. The proposed FRDF enables implicit type conversion from the video frame type to the macro block type at the input arc of the ME block (Figure 2(b)). The fraction rate 1/99 indicates that the macro block is 1/99 of the video frame in its size. How to divide the fraction depends the frame type and should be specified by the programmer before run time. For each execution of the ME block, it consumes a macro-block from the input frame, and produces a macro-block output. Thus, we do not need the D block any more since type conversion is already made inside the ME block. A possible schedule becomes 99(ME,EN). Now, the buffer size on arc α becomes the macro-block size. Formal definition and static analysis of FRDF model is omitted due to space limitation[9].

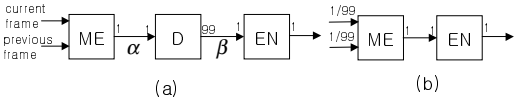

Figure 2. A subgraph of an H.263 encoder example: (a) SDF representation (b) FRDF representation

3. Buffer Sharing Technique

To generate a code from the given FRDF graph, the block execution schedule is determined at compile time. In the buffer sharing technique, global buffers store the live data samples of composite type while the local pointer buffers store the pointers to the global buffer entries. Since multiple data samples can share the buffer space as long as their lifetimes do not overlap, we should examine the lifetimes of data samples. We denote *s(a,k)* as the *k-th* stored sample on arc *a* during an iteration cycle. Consider an example of figure 3(a) with the associated schedule. Within an iteration cycle, two samples are produced and consumed on arc *a*: they are denoted by *s(a,1)* and *s(a,2)* respectively. Arc *b* has an initial sample *s(b,1)* and two more samples, *s(b,2)* and *s(b,3)*, during an iteration cycle.

The lifetimes of data samples are displayed in the *sample lifetime chart* as shown in Figure 3(b) where the horizontal axis indicates the abstract notion of time: each invocation of a node is considered to be one unit of time. The vertical axis indicates the memory size and each rectangle denotes the lifetime interval of a data sample. Note that each sample lifetime defines a single time interval whose start time is the invocation time of the source block and the stop time is after the invocation of the destination block. The lifetime interval of sample *s(b,2)* is [B1, C1]. We take special care of initial samples. The lifetime of sample *s(b,1)* is carried forward from the last iteration cycle while that of sample *s(b,3)* is carried forward to the next iteration cycle. We denote the former-type interval as a *tail lifetime interval*, or shortly a tail interval, and the latter as a *head lifetime interval*, or a head interval. In fact, sample *s(b,3)* at the current iteration cycle becomes *s(b,1)* at the next iteration cycle. To distinguish iteration cycles, we use $s_k(b,2)$ to indicate sample *s(b,2)* at the *k-th* iteration. Then, in Figure 3, $s_1(b,3)$ is equivalent to $s_2(b,1)$. And, the sample lifetime that spans multiple iteration cycles is defined as a *multi-cycle lifetime.*

From the sample lifetime chart, it is obvious that the minimum size of global buffer memory is the maximum of the total memory requirement of live data samples over time. We summarize this fact as the following lemma without proof.

Lemma 1
The minimum size of global buffer memory is equal to the maximum total size of live data samples during an iteration cycle.

We map the sample lifetimes to the global buffers: an example is shown in Figure 3(c) where g(k) indicates the k-th global buffer. In case all data types have the same size, an interval scheduling algorithm can optimally map the sample lifetimes to the minimum size of global buffer memory.

Sample lifetime is distinguished from local buffer lifetime since a local buffer may store multiple samples during an iteration cycle. Consider an example of Figure 3(a) where the local buffer sizes of arcs *a* and *b* are set to be 1 and 2. We denote *B(a,k)* as the k-th local buffer entry on arc *a*. Then, the *local buffer lifetime chart* becomes as drawn in Figure 3(d). Buffer *B(a,1)* stores two samples, *s(a,1)* and *s(a,2)*, to have multiple lifetime intervals during an iteration cycle.

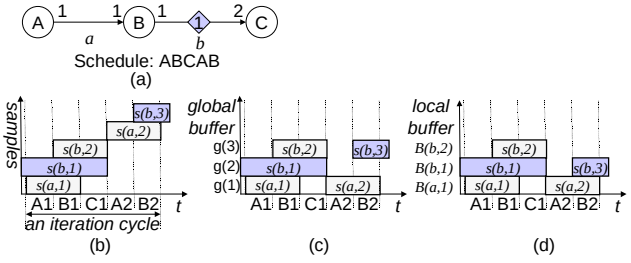

Figure 3. (a) An example SDF graph with an initial delay between B and C illustrated by a diamond (b) the sample lifetime chart (c) the global buffer lifetime chart (d) the local buffer lifetime chart.

Now, we state the problem we have to solve as follows.

Problem:
Determine both the local buffer sizes and the global buffer sizes for the objective of minimizing the sum of them, with the given program graph and the sample lifetime chart that can be obtained from the given schedule.

Since the size of composite data type is usually much larger than that of pointer type in multimedia applications of interest, reducing the global buffer size is more important than reducing the local pointer buffers. Therefore, our sharing heuristic consists of two phases: the first phase is to map the sample lifetimes within an iteration cycle into the minimum size of global buffers ignoring local buffer sizes, and the second phase is to determine the minimum local buffer sizes that store the sample pointers to the given global buffers.

3.1 Global Buffer Minimization

Recall that a sample lifetime has a single interval within an iteration cycle. When all samples have the same data size, the interval scheduling algorithm is known to be an optimal algorithm to find the mapping to the minimum global buffer size. We summarize the interval scheduling algorithm in Figure 4.

1:U is a set of sample lifetimes;
2:P is an empty global buffer lifetime chart.
3:While (U is not empty) {
4: Take out a sample lifetime x with the earliest start time from U.
5: Find out a global buffer whose lifetime ends earlier than the start time of x:
6: Priority is given to the buffer that stores samples on the same arc if exists.
7: If no such global buffer exists in P, create another global buffer.
8: Map x to the selected global buffer
9:}

Figure 4. Interval scheduling algorithm

Consider an example of Figure 3(a), whose global buffer lifetime chart is the same as that is displayed in Figure 3(c). After samples *s(a,1)*, *s(b,1)* and *s(b,2)* are mapped into three global buffers, *s(a,2)* can be mapped to all three buffers. Among the candidate global buffers, we select one that already stores *s(a,1)* according to the policy of line 6 of Figure 4. The reason of this priority selection is to minimize the local buffer sizes, which will be discussed in the next section.

3.2 Local Buffer Size Determination

The global buffer minimization algorithm in the previous phase runs for one iteration cycle while the graph will be executed repeatedly. The next phase is to determine the minimum local buffer sizes that are necessary to store the pointers of data samples mapped to the global buffers. Initially we assign a separate local buffer to each live sample during an iteration cycle. Then, the local buffer size on each arc becomes the total number of live samples within an iteration cycle: Each sample occupies a separate local buffer. In Figure 3(a) for instance, two local buffers are allocated on arc *a* while three local buffers on arc *b*.

What is the optimal local buffer size? The answer depends on when we set the pointer values, or when we *bind* the local buffers to the global buffers. If binding is performed statically at compile-time, we call it *static* binding. If binding can be changed at run-time, it is called *dynamic* binding. In general, the dynamic binding can reduce the local buffer size significantly with run-time overhead of setting pointer values.

3.2.1 Dynamic Binding Strategy

Since we can change the pointer values at run-time in dynamic binding strategy, the local buffer size of an arc can be as small as the maximum number of live samples at any time instance during an iteration cycle.

Consider arc b of Figure 3(a) that has an initial sample and needs only two local buffers since there are at most two live samples at the same time. The global buffer lifetime chart does not repeat itself at the next iteration cycle. Lifetime patterns of local buffers *B(b,1)* and *B(b,2)* are interchanged at the next iteration cycle as shown in Figure 5. We can repeat this pointer assignment at every iteration cycle dynamically. The repetition periods of pointer assignment for arcs with initial samples may span multiple iteration cycles. The next section is devoted to computing the repetition period of pointer assignment for the arcs with initial samples.

Suppose an arc *a* has *N* live samples and *M* local buffers within an iteration cycle. Since the local buffers are accessed sequentially, a local buffer has at most $\lceil N/M \rceil$ samples and the pointer to sample *s(a,k)* is stored in *B*(*a*, *k* mod *M*). After the first phase completes, we examine the mapping results of the allocated sample in a local buffer to the global buffers at the code generation stage. If the mapping result of the current sample is changed from the previous one, we have to insert code to alter the pointer value at the current schedule instance. Note that it incurs both memory overhead of code insertion and time overhead of runtime mapping.

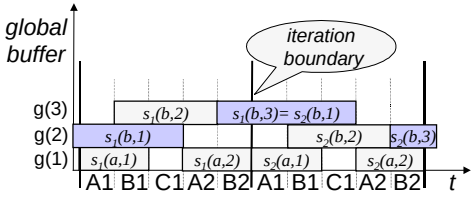

Figure 5. The global buffer lifetime chart spanning two iteration cycles for the example of Figure 3.

3.2.2 Static Binding Strategy

If we use static binding, we may not change the pointer values of local buffers at run time. It means that all allocated samples to a local buffer should be mapped to the same global buffer. For example, arc *a* of Figure 3 needs only one local buffer for static binding since two allocated samples are mapped to the same global buffer. How many buffers do we need for arc *b* of Figure 3 for static binding?

To answer this question, we extend the global buffer lifetime chart over multiple iteration cycles until the sample lifetime patterns on the arc become periodic. We need to extend the lifetime chart over two iteration cycles as displayed in Figure 5. Note that the head lifetime interval of $s_2(b,3)$ is connected to the tail lifetime interval of $s_3(b,1)$ in the next repetition period. Therefore, four live samples are considered during the repetition period that consists of two iteration cycles. The problem is to find the minimum local buffer size *M* such that all allocated samples on each local buffer are mapped to the same global buffer. The minimum number is 4 on arc *b* in this example since $s_3(b,1)$ can be placed at the same local buffer as $s_1(b,1)$.

How many iteration cycles should be extended is an equivalent problem to computing the repetition period of pointer assignment for dynamic binding case. Refer to the next section for detailed discussion.

4. Repetition Period of Sample Lifetime Patterns

Initial samples may make the repetition period of the sample lifetime chart longer than a single iteration cycle since their lifetimes may span to multiple cycles. For simplicity, we assume that all samples have the same size in this section. This assumption will be released in the next section.

First, we compute the iteration length of a sample lifetime. Suppose *d* initial samples stay alive on an arc and *N* samples are newly produced for each iteration cycle. Then, *N* samples on the arc are consumed from the destination node. If *d* is greater than *N*, the newly produced samples all live longer than an iteration cycle. Otherwise, *N-d* newly created samples are consumed during the same iteration cycle while *d* samples live longer. We summarize this fact in the following lemma.

Lemma 2.
If there are *d* initial samples on an arc and *N* samples are newly created per iteration cycle on an arc, the lifetime interval of $d \bmod N$ newly created samples on the arc spans $\lceil d/N \rceil + 1$ iteration cycles and that of $N - (d \bmod N)$ samples spans $\lceil d/N \rceil$ iteration cycles.

Let *p* be the number of iteration cycles in which a sample lifetime interval lies. Figure 6 illustrates two patterns that a sample lifetime interval can have. A sample starts its lifetime at the first iteration cycle with a head interval and ends its lifetime at the *p-th* iteration with a tail interval. Note that the tail interval at the *p-th* iteration also appears at the first iteration cycle. The first pattern, as shown in Figure 6(a), occurs when the tail interval is mapped to the same global buffer as the head interval. The interval mapping pattern repeats every *p-1* iteration cycles in this case.

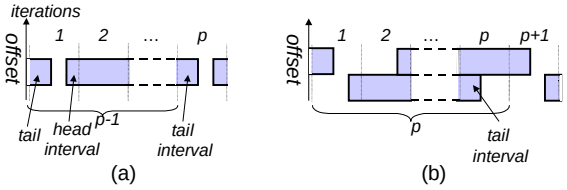

Figure 6. Illustration of a sample lifetime interval: (a) when the tail interval is mapped to the same global buffer as the head lifetime interval, and (b) when the tail interval is mapped to a different global buffer and there is no chained multi-cycle sample lifetime interval.

The second pattern appears when the tail interval is mapped to a different global buffer. To compute the repetition period, we have to examine when a new head interval can be placed at the same global buffer. Figure 6 (b) shows a simple case that a new head interval can be placed at the next iteration cycle. Then, the repetition period of the sample lifetime pattern becomes *p*. More general case occurs when another multi-cycle sample lifetime on a different arc is chained after the tail interval. A multi-cycle lifetime is called *chained* to a tail interval when its head interval is placed at the same global buffer. The next theorem concerns this general case. Due to space limitation, we skip the proof of the theorem.

Theorem 2. Let t_i be the tail interval and h_i the head interval of sample *i* respectively. Assume the lifetime of sample *i* spans p_i *+1* and t_i is chained to the lifetime of sample *i+1* for *i = 1* to *n-1*. The interval mapping pattern repeats every $\sum_{i=1}^{n} p_i$ iteration cycles if interval t_n is chained back to the lifetime of sample *1*. Otherwise it repeats every $\sum_{i=1}^{n} p_i + 1$ iteration cycles.

We apply the above theorem to the case of Figure 3 (b) where head interval *s(b,3)* and tail interval *s(b,1)* are mapped to the different global buffers. And, the sample lifetime spans 2 iteration cycles. Therefore, the repetition period becomes 2 and Figure 5 confirms it.

Another example graph is shown in Figure 7(a), which is identical to the simplified H.263 encoder example of Figure 1. There is a delay symbol on arc CA with a number inside which indicates that there is an initial sample, *s(c,1)*. Assume that the execution order is ABC. During an iteration cycle, sample *s(c,1)* is consumed by A and a new sample *s(c,2)* is produced by C as shown in Figure 7(b). By interval scheduling, an optimal mapping is found like Figure 7(c). By theorem 2, the mapping patterns of *s(c,1)* and *s(c,2)* repeat every other iteration cycles since head interval *s(c,2)* is not mapped to the same global buffer as tail interval *s(c,1)*. Initial samples also affect the lifetime patterns of samples on the other arcs if they are mapped to the same global buffers as the initial samples are mapped to. In Figure 7(c), sample *s(b,1)* are mapped to the same global buffer with *s(c,1)* while *s(a,1)* with *s(c,2)*. As a result, their lifetime patterns also repeat themselves every other iteration cycles. The summary of repetition periods is displayed in Figure 7(d).

Recall that the repetition periods determine the period of pointer update in the generated code with dynamic binding strategy, and the size of local buffers in the generated code with static binding strategy. Figure 7(e) and (f) show the code segments that highlight

the difference. In this example, using dynamic binding strategy is more advantageous.

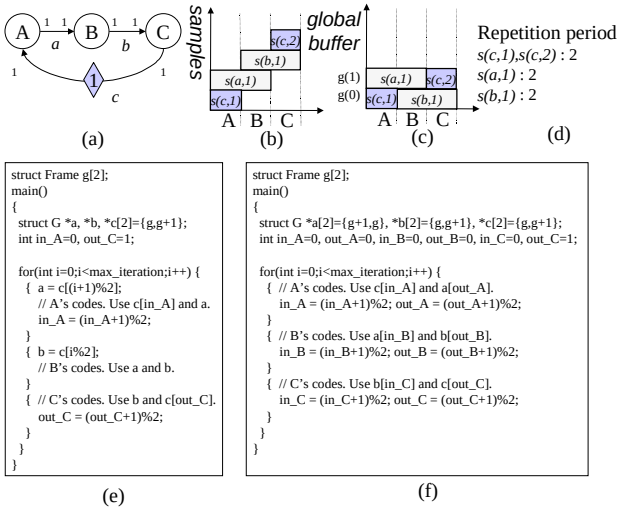

Figure 7. (a) A graph which is equivalent to Figure 1(a) (b) Life-time intervals for an iteration cycle (c) Placement of intervals (d) Determination of repetition period (e) Generated code with dynamic binding (f) with static binding

Up to now, we assume that all samples have the same size. The next section will discuss the extension of the proposed idea to more general case, where samples of different sizes share the same global buffer space.

5. Different Sample Size

We are given sample lifetime intervals which are determined after the scheduled execution order of blocks. The optimal assignment problem of local buffer pointers to the global buffers is nothing but to pack the sample lifetime intervals into a single box of global buffer space. Since the horizontal position of each interval is fixed, we have to determine the vertical position, which is called the "vertical offset" or simply "offset". The bottom of the box, or the bottom of the global buffer space has offset 0. The objective function is to minimize the global buffer space. Recall that if all samples have the same size, interval scheduling algorithm gives the optimal result. Unfortunately, however, the optimal assignment problem with intervals of different sizes is known to be an NP-hard problem. The lower bound is evident from the sample lifetime chart; it is the maximum of the total sample sizes live at any time instance during an iteration. Therefore, we propose a simple but efficient heuristic algorithm. If the graph has no delays (initial samples), we can repeat the assignment every iteration cycle.

The proposed heuristic is called LOES(lowest offset first and earliest start time first). As the name implies, it assigns intervals in the increasing order of offsets, and in the increasing order of start times as a tie-breaker. At the first step, the algorithm chooses an interval that can be assigned to the smallest offset, among unmapped intervals. If more than one intervals are selected, then an interval is chosen which starts no later than others. The earliest start time first policy allows the placement algorithm to produce an optimal result when all samples have the same size since the algorithm is equivalent to the interval scheduling algorithm.

The detailed algorithm is depicted in Figure 8. In this pseudo code, *U* indicates a set of unplaced sample lifetime intervals and *P* a set of placed intervals. At line 5, we compute the feasible offset of each interval in *U* . Set *C* contains intervals whose feasible offsets are lowest among unplaced intervals at line 7. We select the interval with the earliest start time in *C* at line 9 and place it at its feasible offset to remove it from *U* and add it to *P*. This process repeats until every interval in *U* is placed.

```
1: Procedure LOES(U is a set of sample lifetimes) {
2:  P←{}.
3:  While(U is not empty) {
4:    /* compute feasible offsets of every interval in U with P */
5:    computeLowestOffset(U,P);
6:    /* 1st step: choose intervals with the smallest feasible offset
       from U */
7:    C ←findIntervalsWithLowestOffset(U);
8:    /* 2nd step(tie-breaking) : interval scheduling */
9:    select interval x with the earliest arrival time from C;
10:   remove x from U.
11:   P ←P ⋃ {x}.
12:  }
13:}
```

Figure 8. Pseudo code of LOES (Lowest Offset first and Earliest Start-time first) algorithm

If a graph has initial samples and samples have different sizes this method breaks down as Figure 9 illustrates with a simple example. We assume that "*h*" and "*t*" indicate the head interval and the tail interval of the same sample lifetime respectively. At the second iteration, interval *h* should be placed as shown in Figure 9(b) since it is extended from the first cycle. The head interval *h* prohibits interval *x* from lying on contiguous memory space at the second iteration. Such splitting is not allowed in the generated code since the code regards each sample as a unit of assignment. To overcome this difficulty, multi-cycle intervals do not share the global buffer space with other intervals with different sample size. The modified LOES heuristic is omitted due to space limitation.

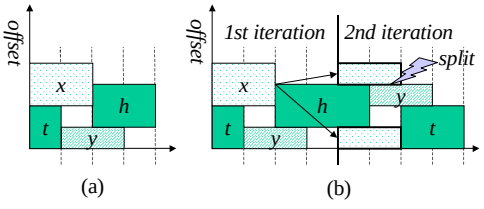

Figure 9. (a) After interval allocation (b) Interval *x* is split at the second iteration

6. Experiments

In this section, we demonstrate the performance of the proposed technique with three real-life examples: a jpeg encoder, an MP3 decoder, and an H.263 encoder. We have implemented the proposed technique in a design environment called PeaCE[10].

Table 1 summarizes the performance improvement from the proposed techniques. The third row indicates the unoptimized buffer requirement in the synthesized code from the initial SDF graph. The fourth row denotes the case of buffer sharing of same size samples without separating local and global buffers. It corresponds to the results expected from the previous works. The fifth and the last rows show the results from the proposed technique of buffer sharing of different size samples without and with buffer separation respectively.

A jpeg encoder example represents the first and the simplest class of program graphs in which all non-primitive data samples have the same size and no initial delay exists. In this case, we do not have to separate the local pointer buffer and the global data buffer, which is taken into account in the proposed technique. As a result, we can reduce the memory requirements to one third as the third and the fourth rows of Table 1 illustrate.

An MP3 decode example is composed of three kinds of different size samples. It represents the second class of graphs that have different size samples but no initial delay sample. No separation between local and global buffers is necessary because there is no initial delay in this example. However, the proposed algorithm that shares the buffer space between different size samples reduces the memory requirement by 52% compared with any sharing algorithm that shares the buffer space among only equal size samples. The fourth and the fifth rows show the performance difference.

As for an H.263 encoder example, we make two versions: the simplified version as discussed in the first section and the full version. The simplified version is an example of the third class of graphs in which all non-primitive data samples have the equal size and initial delay samples exist. As discussed earlier, separation of local buffers from global buffers allows us to reduce the buffer space to optimum as the sixth row reveals. The full version of an H.263 encoder example represents the fourth and the most general class of graphs that consist of different size samples and initial samples. The H.263 encoder example include 4 different size sample sizes and 8 initial delay samples on eight different arcs. The proposed technique can reduce the memory requirement by 40% compared with the unshared version. On the other hand, a sharing technique reduces the buffer size only 23% if both buffer separation and sharing between different samples are not considered.

Table 1. Comparison of buffer memory requirements

Example	JPEG	MP3	Simplified H.263	H.263
# of samples	6	336	3	1804
no sharing	1536B	36KB	111KB	659KB
sharing of same size	512B	23KB	111KB	510KB
sharing w/o separation	512B	11KB	111KB	510KB
sharing with separation	-	-	74KB	396KB

Table 2. Comparison of synthesize codes with reference code for the H.263 encoder in FRDF

Example	Ref. Code	FRDF only	FRDF & buffer sharing
H.263	350KB	225KB	219KB

The SDF model has a limitation that it regards a sample of non-primitive type as a unit of data delivery. In an H.263 encoder example, the SDF model needs an additional block that explicitly divides a frame into 99 macro blocks, paying non-negligible data copy overhead and extra frame-size buffer space. In the manually written reference code, such data copy is replaced with pointer operation. Therefore, we apply the proposed technique to an extended SDF model, called fractional rate dataflow (FRDF). With the FRDF model, we could remove such data copy overhead. And, the proposed buffer sharing technique further reduces the memory requirement by 37% more than the reference code.

7. Conclusion

In this paper, we present efficient code synthesis techniques from dataflow graphs for multimedia applications. Since multimedia applications require large size buffers containing composite type data, we propose two techniques to minimize data buffer size: fractional rate dataflow(FRDF) extension and buffer sharing based on local and global buffer separation.

Through the proposed techniques, automatic software synthesis from a dataflow program graph achieves the comparable code quality with the manually optimized code in terms of memory requirement, which is demonstrated by experiments with three real-life examples.

In this paper, we assume that the execution order of blocks is given from the compile-time scheduling. In the future, we will develop an efficient scheduling algorithm which minimizes the memory requirement based on the proposed technique.

8. REFERENCES

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

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

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

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

[5] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "APGAN and RPMC: Complementary Heuristics for Translating DSP Block Diagrams into Efficient Software Implementations", DAES, vol. 2, no. 1, pp. 33-60, January, 1997.

[6] P. K. Murthy, and S. S. Bhattacharyya, "Shared Buffer Implementations of Signal Processing Systems Using Lifetime Analysis Techniques", IEEE TCAD, vol. 20, no. 2, Feb., 2001.

[7] S. Ritz, M. Willems, H. Meyr, "Scheduling for Optimum Data Memory Compaction in Block Diagram Oriented Software Synthesis", Proceedings of ICASSP, p.2651-2653, 1995

[8] E. D. Greef, F. Catthoor, and H. D. Man, "Array Placement for Storage Size Reduction in Embedded Multimedia Systems", Proceedings of the International Conference on Application-Specific Array Processors, pp. 66-75, July 1997.

[9] H. Oh and S. Ha, "Fractional Rate Dataflow Model and Efficient Code Synthesis for Multimedia Applications," accepted by LCTES/SCOPES '02, June, 2002.

[10] http://peace.snu.ac.kr/research/peace, PeaCE : Codesign Environment.