DATA MEMORY MINIMIZATION BY SHARING LARGE SIZE BUFFERS

Hyunok Oh Soonhoi Ha
The Department of Computer Engineering
Seoul National University
KOREA
{oho,sha}@comp.snu.ac.kr

Abstract - This paper presents software synthesis techniques to deal with non-primitive data type from graphical dataflow programs based on the synchronous dataflow (SDF) model. Non-primitive data types, often used in multimedia and graphics applications, require buffer memory of large size. To minimize the buffer requirement, we separate global data buffers and local pointer buffers. The proposed approach first allocates the minimum size of global buffers and next binds the local buffers to the global buffers by setting the pointers. Static binding and dynamic binding techniques are devised. Experimental results prove the significance of the proposed techniques.

I. Introduction

Reducing the size of memory is an important objective in the embedded system design since an embedded system has tight area and power budgets. Therefore, application designers usually spend significant portion of code development time to optimize the memory requirements.

On the other hand, as system complexity increases and fast design turn-around time becomes important, it attracts more attention to use high level software design methodology: automatic code generation from block diagram specification. COSSAP[1], GRAPE[2], and Ptolemy[3] are well-known design environments, especially for digital signal processing applications, with automatic code synthesis facility from graphical dataflow programs. This paper is concerned with memory-optimized code synthesis from dataflow programs in case applications deal with non-primitive data types such as arrays and structure data types.

An example block diagram representation with dataflow semantics is shown in figure 1. A block represents a function module that consumes data samples from all input arcs and produces data samples to all output arcs. When generating a code from a dataflow graph, a buffer is allocated to each arc to store the data samples between the source and the destination blocks. The allocated buffer size should be no less than the product of the maximum number of samples accumulated on the arc and the size of data type.

This research is supported for Scientific Research 98-E-2103 from the Ministry of Education, Korea.

An application requires memory for both code segments and data segments that store not only data samples but also constants and parameters. If an application handles only data samples of primitive type, such as integer and float, the buffer requirements usually take only a small part of the total memory requirements. However, there are several classes of applications that deal with non-primitive data types. The natural data type of an image processing application is a matrix of fixed block size as illustrated in figure 1. Graphic applications usually need to deal with structure-type samples that contain informations on vertex coordinates, view points, light sources, and so on. Networked multimedia applications may need to exchange packets of data samples between blocks. In those applications, the buffer requirements are likely to be the dominant memory hog.

Once the execution order of blocks is determined, the buffer requirements on all arcs as well as the code size are also determined. Thus, the problem of minimizing memory requirements has been considered as a scheduling problem to determine the execution order of blocks with the objective of memory minimization[4][5]. In this paper, we introduce two buffer sharing techniques to further reduce the buffer memory requirements with a given scheduling result. Since most existent schedulers do not consider the buffer sharing of non-primitive type data, the proposed optimization techniques are complementary to their previous efforts of memory minimization.

Figure 1 shows a demonstration how much data memory we can reduce by sharing buffers. Without buffer sharing, 5 blocks whose size is 64(=8x8) are needed. However, only two blocks are needed if buffer sharing is used since a, c and e buffers share A buffer, and b and d buffers share B buffer.

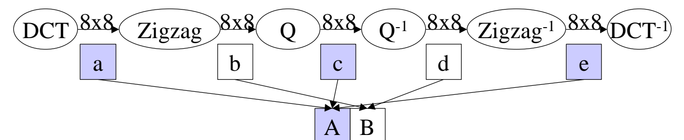


Figure 1. Image processing example

In this paper, we discuss how to assign as small buffers as possible to store non-primitive type data samples. To this end, we separate global data buffers and local pointer buffers as illustrated in figure 1. A and B buffers created by memory allocation, store data samples while arcs are associated with

pointer buffers a, b, c, d and e that will be bound to A or B buffer.

We explain the problem of memory optimized code synthesis and review the related works in the next section. In section 3, a new buffer sharing problem is defined with the non-primitive data types. In sections 4 and 5, we present our optimization techniques based on static binding and dynamic binding respectively. We show some experiments and conclude the paper afterwards.

II. Memory Optimized Code Synthesis with Primitive Data Types

We use synchronous dataflow (SDF) model[6] as the block diagram semantics while the proposed techniques are applicable to other dataflow models of computation. In an SDF graph, a block may consume or produce multiple number of samples per execution. Each arc is annotated with the number of samples produced or consumed by an invocation of its source or destination nodes. We illustrate an example SDF graph in figure 2(a).

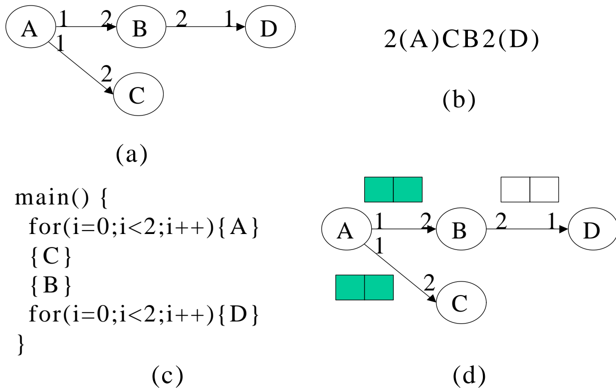


Figure 2. (a) SDF graph example (b) Scheduling result (c) Code generation (d) Buffer allocation and binding

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

The generated code template according to the schedule of figure 2(b) is shown in figure 2(c). Note that the code of each block appears only once even though the schedule contains two invocations of block A and D. If we generate the code starting from another valid schedule "2(A)BDCD", the code of block D would appear twice. Hence, the schedule of figure 2(b) is called a "single appearance schedule (SA-schedule)"[4] meaning that the code of each block appears only once in the generated code. An SA-schedule implies that the code size of the generated code from the given dataflow graph is minimal.

Now, let us examine the buffer requirement. After block A is executed twice, two data samples are produced on each output arc as explicitly depicted in figure 2(d). We define the buffer allocated on each arc as a *local* buffer that is used for local communication between two associated blocks. If the data samples are of primitive types, the local buffers store data values and the generated code defines the local buffers as arrays of primitive types. Suppose all data samples are of the same type in the example graph, there is a possibility of buffer sharing in the schedule of figure 2(b). The local buffer between blocks A and C can be reused as the local buffer between blocks B and D since the life-times of data samples stored in those buffers do not overlap. If we consider the buffer sharing possibility, the total size of local buffers is reduced to 4 from 6 without buffer sharing.

To generate the code with minimal memory requirement, both the code size and the local buffer size should be minimized at the same time. Most previous approaches considered single appearance schedules to minimize the code size first. Bhattacharyya et al. developed two clustering heuristics, APGAN and RPMC, to find out a single appearance schedule with minimal local buffers, but ignoring the possibility of buffer sharing[4]. On the other hand, Ritz et. al. used an ILP formulation to find out a "flat"(which means that the looping of nodes is not hierarchical) single appearance schedule with minimum local buffer sizes, considering buffer sharing[5]. A flat SA-schedule does not allow nested looped schedule. Since it usually requires more local buffers than the optimal nested SA-schedule, it is not evident in general whether the advantage of buffer sharing outweighs the limitation of flat SA-schedules.

III. Buffer Sharing Problem with Non-primitive Data Types

It is well known that the naive implementation of dataflow model may incur a large overhead for handling data samples of non-primitive data type. Suppose that each local buffer stores a matrix sample. If a block consumes an input matrix sample, modifies an element of the matrix, and produces an output matrix sample, even the other unmodified elements should be copied from the input local buffer to the output local buffer. To avoid such overhead, we distinguish *global* buffers from local buffers. A local buffer contains a pointer to a global buffer that stores a real matrix data sample. Figure 3(a) illustrates the buffer structure consisting of separate local and global buffers with the example of figure 2(a). It is assumed that the data types are non-primitive. Since the size of pointer is comparable to that of primitive-type, global buffer is not needed for primitive data types as has been mainly assumed in the previous researches. Note that a

global buffer can be bound to multiple local buffers as the adjunct "global" indicates. By making the input and the output local buffers point the same global buffer, we can avoid redundant copy of unmodified elements.

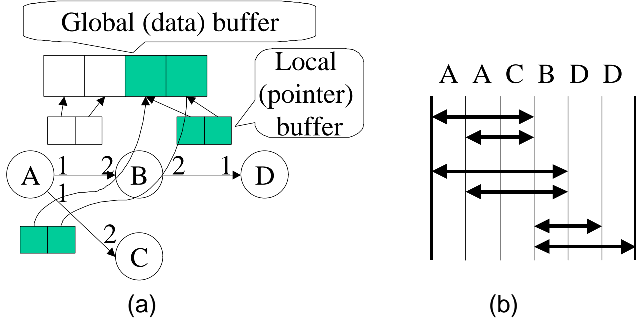


Figure 3. (a) Global data buffer and local pointer buffer (b) Local buffer life-time chart

With the proposed buffer structure of separate local and global buffers, we also break buffer management task into two sub-tasks: buffer allocation and buffer binding. Through compile-time analysis based on the given schedule, we determine the size of local buffers and global buffers statically, which defines the buffer allocation sub-task. Buffer binding is to set the pointer value of a local buffer to an available global buffer. Depending on when this buffer binding actually happens, *static binding* and *dynamic binding* are classified. In the static binding strategy, the pointer values of local buffers are predetermined at compile-time. On the other hand, the dynamic binding strategy set the pointer values at run-time.

Now, we have to minimize the size of global buffers as well as local buffers. Previous works tried to minimize the local buffers only[4][5]. With non-primitive data types, however, minimizing global buffers is usually more important. If we bind each local buffer to a separate global buffer throughout the whole schedule, minimizing the local buffer size is equivalent to minimizing the global buffer size. Unfortunately, such a trivial binding is not always optimal as will be shown later.

The problem of minimizing the total size of local buffers and global buffers with static binding is NP-hard. Therefore, our approach is to minimize the global buffer first since the global buffer size would be more significant in case of non-primitive type data samples. The minimum size is nothing but the maximum number of live local buffers during an iteration of the given schedule. Figure 3(b) displays *the local buffer life-time chart* in which the *x*-axis represents buffer life-time whose unit is the execution of each node and the *y*-axis indicates each local buffer. From the life-time chart, the global buffer size is determined to 4 which is the maximum number of local buffers whose life times are overlapped at any instant. With the minimum global buffers, we introduce a static binding and a dynamic binding technique. In the static binding technique, we give up minimizing local buffers to avoid run-time overhead of memory management. On the other hand, the dynamic binding technique uses the minimal local buffers, but binds them to the global buffers at run-time.

For the comparison purpose, we display the generated code from figure 1 with the traditional approach without global buffers and buffer sharing in figure 4.

```
struct Block8x8 { int pixel[8][8]; };
/* main function */
int main(int argc, char *argv[]) {
struct Block8x8* output_0;
struct Block8x8* output_1;
struct Block8x8* output_2;
struct Block8x8* output_3;
struct Block8x8* output_4;

output_0=(void*)malloc(sizeof(struct Block8x8));
output_1=(void*)malloc(sizeof(struct Block8x8));
…
{ int sdfLoopCounter_5;for (sdfLoopCounter_5 = 0; sdfLoopCounter_5
< 10; sdfLoopCounter_5++)
…
}}
```

Local buffer

Buffer allocation

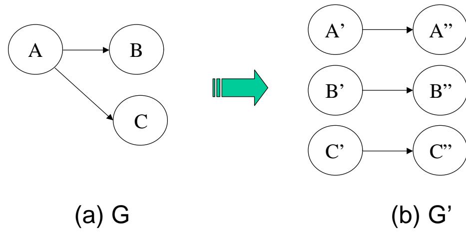
Figure 4. Code generation without buffer sharing from figure 1

IV. Static Buffer Binding

As mentioned above, the static buffer binding technique binds a local buffer to a global buffer statically at the variable initialization step. Many local buffers can be connected to one global buffer as long as their life-times do not overlap each other. However, the general buffer sharing problem is NP-hard as stated in the following theorem.

[Theorem] If the life-time of a local buffer consists of multiple time intervals, the decision problem whether there exists a static binding from the given number of local buffers to the global buffers is NP-complete.

(Proof) Consider a graph coloring problem with a graph G=(V,E) where V is a vertex set and E is an edge set. A simple example graph is shown in figure 5(a). We associate a new graph G' (figure 5 (b)) where two nodes are created for each vertex in graph G. The local buffer between two nodes in graph G' is mapped to a vertex in graph G. Note that if we take a valid schedule A'B'A"B", two local buffers may not be shared. For each arc in graph G, we generate a schedule that does not allow buffer sharing. For example, arc(AC) in graph G generates a schedule (A'C'A"C") in graph G'. Now, the valid schedule of graph G' associated with graph G is A'B'A"B"A'C'A"C", in which the life-time of a local buffer on arc A'A" has two intervals. Likewise a graph coloring problem can be reduced to a static binding problem allowing multiple time intervals. Thus the proof completes.



(a) G (b) G'

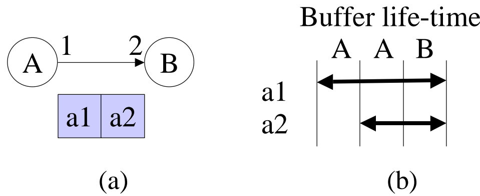
Figure 5. Transformation from (a) a graph coloring problem to (b) a buffer sharing problem

Even if a general buffer sharing problem is NP hard, we can find an optimal static binding by an interval scheduling algorithm if the life-times of all local buffers have a single time interval. The interval scheduling algorithm sorts local buffers in the increasing order of interval starting times and binds one at a time to an available global buffer. If the interval of a local buffer elapses, the mapped global buffer is returned to the available global buffer pool.

The proposed technique is to increase the local buffer size if necessary so that there exists a static binding from the local buffers to the minimum size of global buffers.

Case 1) The graph has no sample delay.

In this case, the local buffer size becomes smaller than or equal to the total number of consuming samples by the destination block during one iteration. In the latter situation, as illustrated in figure 6(a), the life-times of all local buffers consist of only one interval. Thus, we can find an optimal binding by interval scheduling.



(a) (b)

Figure 6. (a) Local buffers (b) Their life-time chart

If the local buffer size is less than the number of consuming samples, the local buffer has multiple intervals. For example, if the number of local buffers between A and B in figure 7 is 6, the life-times of buffers consists of multiple intervals as shown in figure 7(a). To apply the interval scheduling algorithm, we extend the local buffer size to the number of consuming samples, which is 12 in this example (figure 7(b)).

Case 2) The graph has sample delays.

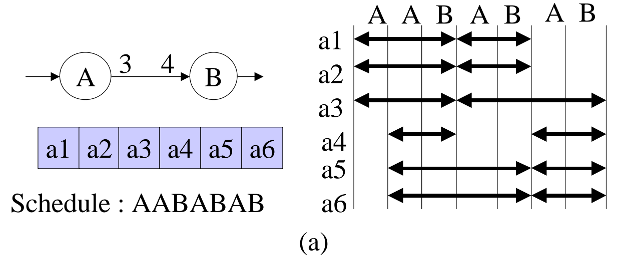
If a local buffer has sample delays then the local buffer size should be greater than the number of consuming samples in order not to have multiple intervals. Consider an example in figure 8(a), where local buffer a2 has 2-intervals over an iteration and the number of global buffers is 3 (figure 8(b)). It is assumed that the initial samples are placed at the end of the local buffer. In this case, we have to increase the local buffer size for interval scheduling. If the local buffer size is a multiple of the number of consuming samples, the interval patterns repeat after some iterations as displayed in figure 8(c). In general, the local buffer size to avoid multiple intervals in one iteration is as follows.

The local buffer size = $(\lceil D_i / N_i \rceil + 1) * N_i$

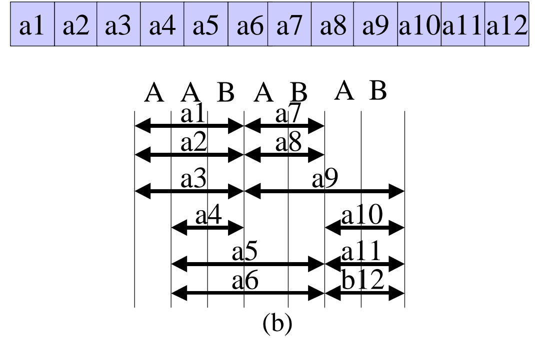
where D_i is the number of delay samples and N_i is the number of consuming samples. In this example,$(\lceil 1/2 \rceil + 1)*2 = 4$.

After determining the local buffer size of each arc, we may examine the buffer life-time chart over as many iterations as the interval pattern is repeated (figure 8(c)). Note that the interval patterns over iteration cycles are permutations to each other. The pattern of a2 and a4 at the first iteration is equal to that of a4 and a2 at the second. Therefore it is sufficient to consider only one iteration.

After interval scheduling over the first iteration, the global buffer sharing result is obtained as figure 8(d). From the interval scheduling result, we can calculate the overall buffer binding result like figure 8(e). Now, each row of the chart represents a set of local buffers, or a sharing pattern of local buffers, mapped to a global buffer. We expand the interval scheduling result until the sharing pattern is repeated (figure 8(e)). This step is the main step to determine the local buffer sizes and static binding. Some local buffers may need to be increased or can be decreased. In this example, the local buffer size of arc AB can be decreased to 2 from 4 if we observe that buffers {a1, a2} can be reused for buffers {a3, a4}. Then the interval pattern of the local buffers on arc AB is repeated after one iteration. In fact, the local buffer size of an arc is equal to the product of the number of samples and the sharing pattern of the bound global buffers. Mathemati-



(a)



(b)

Figure 7. (a) Local buffers have two intervals (b) All local buffers have one interval after increasing the local buffer size

cal analysis is somewhat complicated, thus omitted due to space limitation.

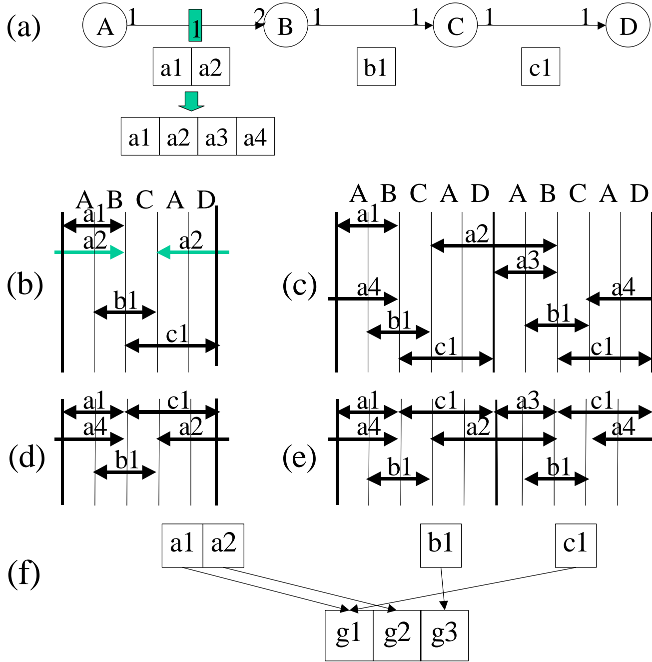


Figure 8. (a) An example graph with sample delays (b) Life-time chart with multiple intervals (c) Life-time chart over two iterations after increasing the local buffer size of arc AB (d) Interval scheduling result (e) Expanding the sharing result (f) Final result of static binding

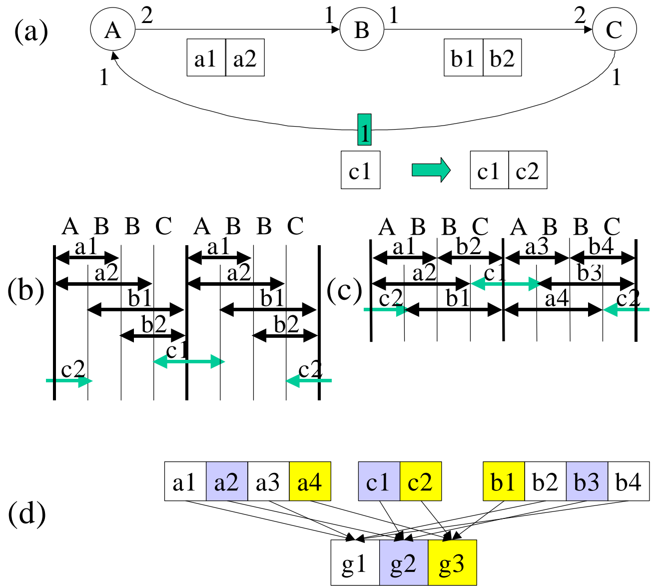


Figure 9. (a) An example graph with a feedback arc (b) Life-time chart (c) After interval scheduling (d) Increasing the local buffer size and binding the local buffers to the global buffers

The other example is shown in figure 9(a). This graph is a simplified version of algorithms with a feedback arc. To use the interval scheduling algorithm, the local buffer size on arc CA is increased to 2. The life-time chart after interval scheduling shown in figure 9(c) represents that a2 and c1 share a global buffer, and c2 and b1 share the other global buffer. Since the interval pattern of the global buffers bound with a2 and b1 are repeated after two iterations, we increase the local buffer sizes of arc AB and arc BC twice.

V. Dynamic Buffer Binding

At the cost of local buffer sizes, the static binding technique could bind the local buffers to the global buffers by the interval scheduling algorithm. In this section, we introduce a dynamic buffer binding technique that does not increase any local buffer while sharing the minimum global buffer size. However we have to pay run-time overhead to bind local buffers dynamically. Figure 10 represents the binding result from figure 9(a). Since the dynamic binding method binds a local buffer to a global buffer dynamically whenever the local buffer is used, it is not needed that the interval pattern of each global buffer is repeated and the local buffer sizes are increased.

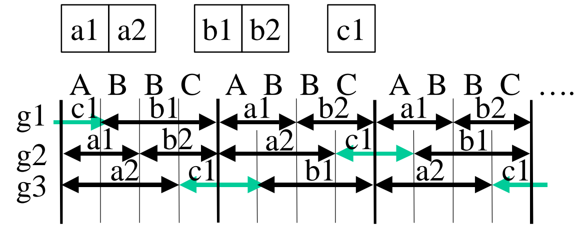


Figure 10. The dynamic binding result from figure 9(a)

Figure 11 displays the code that uses the dynamic binding technique. It contains two functions: p_allocBuffer() to fetch a free global buffer and p_freeBuffer() to return the used global buffer to the free list. Before executing a block, the local buffers on the output arcs are bound to the global buffers and the bound global buffers of the input arcs are released. Binding a local buffer to a global buffer executes two instructions and releasing a global buffer requires 4 or 5 instructions.

Besides the increase of the code size, we also need extra data structure to manage the free list of global buffers. The size of the free list is the same as the number of global buffer entries.

VI. Experiment

The table 1 shows the summary of itemized buffer requirements of three strategies discussed in this paper with the simple example graph of figure 1. As expected, the size of global buffer is reduced significantly with the proposed buffer sharing techniques. Since the graph belongs to case 1 category in section 4, the static binding does not need to increase any local buffers. While the static binding technique needs one extra memory to maintain the global buffer pointer, the dynamic binding technique needs 5 more words

```
struct Block8x8 _output_0_[2];
int p_freeList__output_0_[2] = {1,-1};
int p_freeList_head__output_0_ = 0;
int p_freeList_tail__output_0_ = 1;
int p_allocBuffer(int* theFreeList,int* theHead) {
        int buffer = *theHead;
        *theHead = theFreeList[*theHead];
        return buffer; }
void p_freeBuffer(int theBuffer, int* theFreeList, int* theHead,
int*theTail) {
        theFreeList[*theTail] = theBuffer;
        *theTail = theBuffer;
        theFreeList[*theTail] = -1;
        if(*theHead==-1) *theHead = theBuffer; }
/* main function */
int main(int argc, char *argv[]) {
struct Block8x8* output_0;
…
{ int sdfLoopCounter_5;for (sdfLoopCounter_5 = 0; sdfLoopCounter_5
< 10; sdfLoopCounter_5++) {
output_0=&_output_0_[p_allocBuffer(p_freeList__output_0_,
&p_freeList_head__output_0_)];
        { /*DCT*/ ..}
output_1=&_output_0_[p_allocBuffer(p_freeList__output_0_,
&p_freeList_head__output_0_)];
p_freeBuffer(output_0,_output_0_,p_freeList__output_0_,&p_freeList
_head__output_0_,&p_freeList_tail__output_0_);
        { /* Zigzag*/ ..}.
        ..}}
```


Figure 11. Dynamic buffer binding

(2 for the free list, 2 for the head and tail pointers of the list, and 1 for the global buffer pointer). The dynamic binding technique also takes 30 instructions of run-time overhead per iteration.

Table 1. The buffer sharing result of figure 1

	Global buffer	Local buffer	Total (in words)	Run-time over-head (# of inst's)
Without sharing	320	5	325	0
Static	128	5	134	0
Dynamic	128	5	138	30

We experiment a large real application, H.263 encoder that is an image compression standard for videophones. In reality, figure 1 is a subgraph of the H.263 encoder. In this example, we can reduce the total buffer size by about 40%. It is noteworthy that the dynamic binding requires the least amount of memory with minimum local buffer sizes.

Table 2. The buffer sharing result of H.263

	Global buffer	Local buffer	Total (in words)	Run-time over-head (# of inst's)
Without sharing	444224	2400	446624	0
Static	260288	4370	264663	0
Dynamic	260288	2400	263703	14400

VII. Conclusions

This paper presents software synthesis techniques to deal with non-primitive data type from graphical dataflow programs based on the synchronous dataflow (SDF) model. To minimize the buffer requirements for non-primitive data types, we separate global data buffers and local pointer buffers. The proposed approach first allocates the minimum global buffer size and next binds the local buffers to the global buffers by static binding or dynamic binding. In two experiments, we could reduce 40% and 60% of the global buffer size through buffer sharing.

While we share buffers of the same type in this paper, we will extend this work to share buffers of different types in the future. Furthermore, we will develop an efficient scheduling algorithm to consider global buffer sharing.

References

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

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

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

[4] 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

[5] S. Ritz, M. Willems, H. Meyr, "Scheduling for Optimum Data Memory Compaction in Block Diagram Oriented Software Synthesis", Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, p.2651-2653, 1995

[6] E. A. Lee and D. G.Messerschimitt, "Synchronous Data Flow", Proceedings of the IEEE, vol. 75, no. 9, pp. 1235-1245, 1987