EURASIP Journal on Applied Signal Processing 2003:6, 514–529
© 2003 Hindawi Publishing Corporation

Memory-Optimized Software Synthesis from Dataflow Program Graphs with Large Size Data Samples

Hyunok Oh
The School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-742, Korea
Email: oho@comp.snu.ac.kr

Soonhoi Ha
The School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-742, Korea
Email: sha@comp.snu.ac.kr

Received 28 February 2002 and in revised form 15 October 2002

In multimedia and graphics applications, data samples of nonprimitive type require significant amount of buffer memory. This paper addresses the problem of minimizing the buffer memory requirement for such applications in embedded software synthesis from graphical dataflow programs based on the synchronous dataflow (SDF) model with the given execution order of nodes. We propose a memory minimization technique that separates global memory buffers from local pointer buffers: the global buffers store live data samples and the local buffers store the pointers to the global buffer entries. The proposed algorithm reduces 67% memory for a JPEG encoder, 40% for an H.263 encoder compared with unshared versions, and 22% compared with the previous sharing algorithm for the H.263 encoder. Through extensive buffer sharing optimization, we believe that automatic software synthesis from dataflow program graphs achieves the comparable code quality with the manually optimized code in terms of memory requirement.

Keywords and phrases: software synthesis, memory optimization, multimedia, dataflow.

1. INTRODUCTION

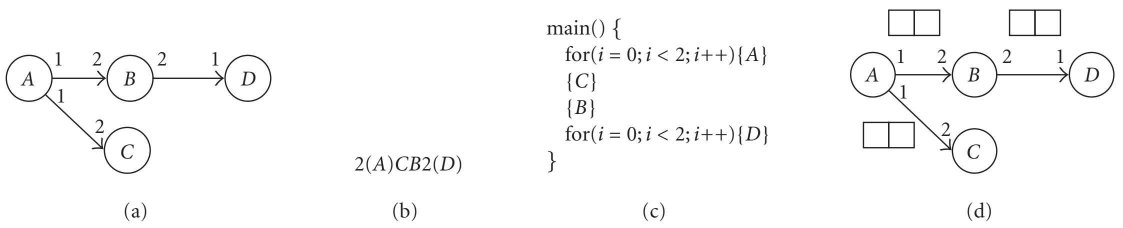
Reducing the size of memory is an important objective in embedded system design since an embedded system has tight area and power budgets. Therefore, application designers usually spend significant amount of code development time to optimize the memory requirement.

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.

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] which is widely adopted in aforementioned design environments. We illustrate an example of SDF graph in Figure 1a. Each arc is annotated with the number of samples consumed or produced per node execution. In this paper, we are concerned with memory optimized software synthesis from SDF graphs though the proposed techniques can be easily extended to other SDF extensions.

To generate a code from the given SDF graph, the order of block executions is determined at compile time, which is called “scheduling.” Since a dataflow graph specifies only partial orders between blocks, there are usually more than one valid schedule. Figure 1b shows one of many possible scheduling results in a list form, where $2(A)$ means that block A 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 1b is shown in Figure 1c.

When synthesizing software from an SDF graph, a buffer space is allocated to each arc to store the data samples



$2(A)CB2(D)$

```
main() {
  for(i = 0; i < 2; i++){A}
  {C}
  {B}
  for(i = 0; i < 2; i++){D}
}
```

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

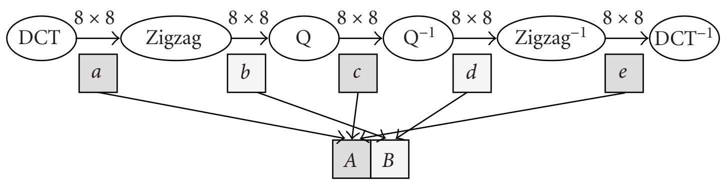


Figure 2: Image processing example.

between the source and the destination blocks. The number of allocated buffer entries should be no less than the maximum number of samples accumulated on the arc at runtime. After block A is executed twice, two data samples are produced on each output arc as explicitly depicted in Figure 1d. We define a buffer allocated on each arc as a local buffer that is used for data transfer between two associated blocks. If the data samples are of primitive types, the local buffers store data values and the generated code defines a local buffer with an array of primitive type data.

Required memory spaces in the synthesized code consist of code segments and data segments. The latter stores constants and parameters as well as data samples. We regard memory space for data samples as buffer memory, or shortly buffer, in this paper.

There are several classes of applications that deal with nonprimitive data types. The typical data type of an image processing application is a matrix of fixed block size as illustrated in Figure 2. Graphic applications usually need to deal with structure-type data samples that contain information on vertex coordinates, viewpoints, light sources, and so on. Networked multimedia applications exchange packets of data samples between blocks. In those applications, the buffer requirements are likely to be more significant than others. For example, the code size of H.263 encoder [5] is about 100 K bytes but the buffer size is more than 300 K bytes.

Since the buffer requirement of an SDF graph depends on the execution order of nodes, there have been several approaches [6, 7, 8] to take the buffer size minimization as one of the scheduling objectives. However, they do not consider either buffer sharing possibilities nor nonprimitive data types. Finding out an optimal schedule for minimum buffer requirements considering both is a future research topic. *In this paper, instead, we propose a buffer sharing technique for nonprimitive type data samples to minimize the buffer memory requirement assuming that the execution order of nodes is already determined at compile time.* Thus, this work is complementary to existent scheduling algorithms to further reduce the buffer requirement.

Figure 2 demonstrates a simple example where we can reduce the significant amount of buffer memory by sharing buffers. Without buffer sharing, five local buffers of size 64 ($= 8 \times 8$) are needed. On the other hand, only two buffers are needed if buffer sharing is used so that a, c, and e buffers share buffer A, and b and d buffers share buffer B. Such sharing decision can be made at compile time through lifetime 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 global data buffers explicitly in the synthesized code. In Figure 2, we use five local pointer buffers and two global buffers. This separation provides more memory sharing chances when the number of local buffer entries becomes more than one. If the local buffer size becomes one after buffer optimization, no separation is needed. We examine Figure 3a which illustrates a simplified version of an H.263 encoder algorithm where "ME" node indicates a motion estimation block, "Trans" is a transform coding block which performs DCT and Quantization, and "InvTrans" performs inverse transform coding and image reconstruction. Each sample between nodes is a frame of 176×144 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 3b since buffers a and c overlap their lifetimes at ME, a and b at Trans, and b and c at InvTrans. Even though two frames are sufficient for this graph, we cannot share any buffer without separation of local buffers and global buffers. In fact, we can use only two frame buffers if we use separate local pointer buffers. Figure 3c shows the allocation of local buffers and global

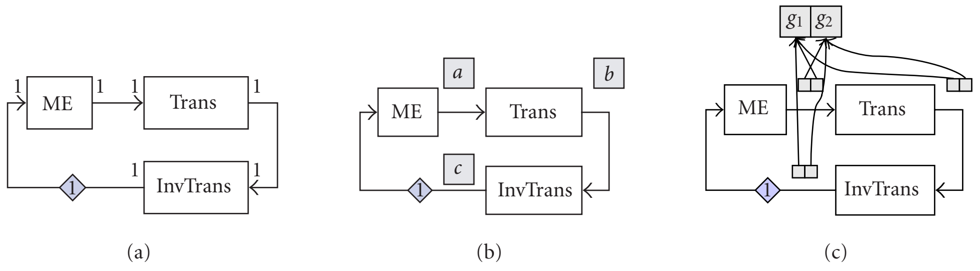


FIGURE 3: (a) Simplified H.263 encoder in which a diamond between InvTrans and ME indicates an initial sample delay, (b) and (c) a minimum buffer allocation without and with separation of global buffers and local buffers, respectively.

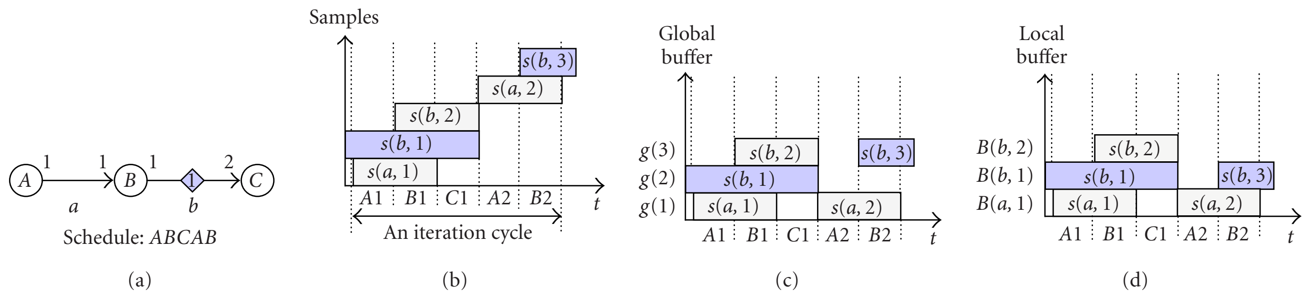


FIGURE 4: (a) An example of SDF graph with an initial delay between B and C illustrated by a diamond, (b) the sample lifetime chart, (c) a global buffer lifetime chart, and (d) a local buffer lifetime chart.

buffers, and the mapping of local buffers to global buffers. The detailed algorithm and code synthesis techniques will be explained in Section 4.

It is NP-hard to determine the optimal local buffer, global buffer sizes, and their mappings in general cases where there are feedback structures in the graph topology. The problem becomes harder if we consider buffer sharing among different size data samples. Therefore, we devise a heuristic that focuses on global buffer minimization first and applies an optimal algorithm next to find the minimum local pointer buffer sizes and to map the local pointer buffers to the minimum global buffers. The proposed heuristic results in less than 5% overhead than an optimal solution on average.

In Section 2, we define a new buffer sharing problem for nonprimitive data types, and survey the previous works briefly. The overview of the proposed technique is presented in Section 3. Section 4 explains how to minimize the size of local buffers and their mappings to the minimum global buffers assuming that all data samples have the same size. In Section 5, we extend the technique to the case where data samples have the different sizes. Graphs with initial samples are discussed in Section 6. Finally, we present some experimental results in Section 7, and make conclusions in Section 8.

2. PROBLEM STATEMENT AND PREVIOUS WORKS

In the proposed technique, global buffers store the live data samples of nonprimitive type while the local pointer buffers store the pointers for 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 kth stored sample on arc a and TNSE(a) as the total number of samples exchanged during an iteration cycle. Consider an example of Figure 4a with the associated schedule. TNSE(a) becomes 2 and two samples, $s(a, 1)$ and $s(a, 2)$, are produced and consumed on arc a. 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 4b, 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 the completion time of the destination block. For example, 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 kth iteration. Then, in Figure 4, $s_1(b, 3)$ is equivalent to $s_2(b, 1)$.

And the sample lifetime that spans multiple iteration cycles is defined as a *multicycle lifetime*. Note that the sample lifetime chart is determined from the schedule.

From the sample lifetime chart, it is obvious that the minimum size of global buffer memory is the maximum of the total memory requirements 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 at any instance during an iteration cycle.*

We map the sample lifetimes to the global buffers: an example is shown in Figure 4c where $g(k)$ indicates the kth global buffer. In case all data types have the same size, an interval scheduling algorithm can successfully 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 4a where the local buffer sizes of arcs a and b are set to be 1 and 2, respectively. We denote $B(a, k)$ as the kth local buffer entry on arc a. Then, the *local buffer lifetime chart* becomes as drawn in Figure 4d. Buffer $B(a, 1)$ stores two samples, $s(a, 1)$ and $s(a, 2)$, to have multiple lifetime intervals during an iteration cycle. Now, we state the problem this paper aims to solve as follows.

Problem 1. Determine $\mathrm{LB}(g, s(g))$ and $\mathrm{GB}(g, s(g))$ in order to minimize the sum of them, where $\mathrm{LB}(g, s(g))$ is the sum of local buffer sizes on all arcs and $\mathrm{GB}(g, s(g))$ is the global buffer size with a given graph g and a given schedule $s(g)$.

Since the simpler problems are NP-hard, this problem is NP-hard, too. Consider a special case when all samples have the same type or the same size. For a given local buffer size, determining the minimum global buffer size is difficult if a local buffer may have multiple lifetime intervals, which is stated in the following theorem.

Theorem 1. *If the lifetime of a local buffer may have multiple lifetime intervals and all data types have the same size, the decision problem whether there exists a mapping from a given number of local buffers to a given number of global buffers is NP-hard.*

Proof. We will prove this theorem by showing that the graph coloring problem can be reduced to this mapping problem. Consider 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 5a. We associate a new graph G' (Figure 5b) where a pair of nodes are created for each vertex of graph G and connected to the dummy source node S and the dummy sink node K of the graph G'. In other words, a vertex in graph G is mapped to a local buffer in graph G'. The next step is to map an arc of graph G to a schedule sequence in graph G'. For instance, an arc AB in graph G is mapped to a schedule segment $(A'B'A''B'')$ to enforce that two local buffers on

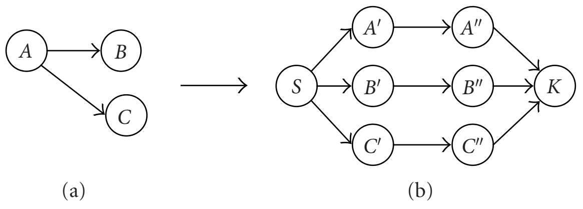


FIGURE 5: (a) An example instance of graph coloring problem, and (b) the mapped graph for the proof of Theorem 1.

arcs $A'A''$ and $B'B''$ may not be shared. As we traverse all arcs of graph G, we generate a valid schedule of graph G'. Traversing arcs AB and AC in graph G generates a schedule: $S(A'B'A''B'')(A'C'A''C'')K$. From this schedule, we find out that the buffer lifetime on arc $A'A''$ consists of two intervals. The constraint that two adjacent nodes in G may not have the same color is translated to the constraint that two local buffers may not be shared in G'. Therefore, the graph coloring problem for graph G is reduced to the mapping problem for graph G'. □

The register allocation problem in traditional compilers is to share the memory space for the variables of nonoverlapped lifetimes [9]. If the variable sizes are not uniform, the allocation problem, known as the dynamic storage allocation problem [10, 11], is NP-complete. In our context, this problem is equivalent to minimize the global buffer memory ignoring the local buffer sizes and mapping problems.

De Greef et al. [12] presented a systematic procedure to share arrays for multimedia applications in a synthesis tool called ATOMIUM. They analyze lifetimes of array variables during a single iteration trace of a C program and do not consider the case where lifetimes span multiple iteration cycles. If the program is retimed, some variables can be live longer than a single iteration cycle. Another extension we make in the proposed approach is that we consider each array element separately for sharing decision when each array element is of nonprimitive type.

Recently, Murthy and Bhattacharyya [13] proposed a scheduling technique for SDF graphs to optimize the local memory size by buffer sharing. Since they assume only primitive type data, their sharing decision considers array variables as a whole. However, their research result is complementary to our work since the schedule reduces the number of live data samples at runtime, which reduces the global memory size in our framework. They compared their research work with Ritz et al.'s [14] whose schedule pattern does not allow nested loop structure. They showed that nested loop structure may significantly reduce the local memory size.

Even though memory sharing techniques have been researched extensively from compiler optimization to high level synthesis, no previous work has been performed, to the authors' knowledge, to solve the problem we are solving in this paper.

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

Figure 6: Interval scheduling algorithm.

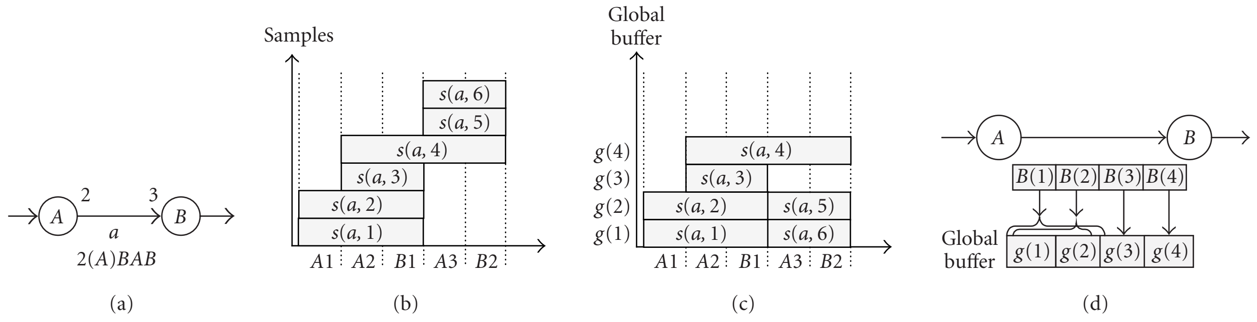


Figure 7: (a) An SDF subgraph with a given schedule, (b) the sample lifetime chart, (c) the global buffer lifetime chart, and (d) local buffer allocation and mapping.

3. PROPOSED TECHNIQUE

In this section, we sketch the proposed heuristic for the problem stated in the previous section. Since the size of nonprimitive 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 heuristic consists of two phases: the first phase is to map the sample lifetimes within an iteration cycle into the minimum number of global buffers ignoring local buffer sizes, and the second phase is to determine the minimum local buffer sizes and to map the local buffers 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 [15] to find the minimum global buffer size. We summarize the interval scheduling algorithm in Figure 6.

Consider an example of Figure 4a whose global buffer lifetime chart is displayed in Figure 4c. 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 5 of Figure 6. The reason of this priority selection is to minimize the local buffer sizes, which will be discussed in the next section.

When the data samples have different sizes, this mapping problem becomes NP-hard since a special case can be reduced to 3-partition problem [10]. Therefore, we develop a heuristic, which will be discussed in Section 5.

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 4a, 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 runtime, it is called *dynamic* binding. In general, the dynamic binding can reduce the local buffer size significantly with small runtime overhead of global buffer management.

3.2.1 Dynamic binding strategy

Since we can change the pointer values at runtime 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 another example of Figure 7a with a given scheduling result and a global buffer lifetime chart as shown in Figure 7c. Since the maximum number of live samples is four, we need at least four local buffers on arc a. Suppose we have the minimum number of local buffers on arc a. Local buffer $B(a, 1)$ stores two samples, $s(a, 1)$ and $s(a, 5)$, which are unfortunately mapped to different global buffers. It means that the pointer value of local buffer $B(a, 1)$ should be set to $g(1)$ at the first invocation of node A but to $g(2)$ at the third invocation, dynamically. We repeat this pointer assignment at every iteration cycle at runtime.

If there are initial samples on an arc, care should be taken to compute the repetition period of pointer assignment. Arc b of Figure 4a has an initial sample and needs only two local buffers since there are at most two live samples at the same time. Unlike the previous example of Figure 7, the global buffer lifetime chart may not repeat itself at the next iteration cycle. The lifetime patterns of local buffers $B(b, 1)$ and $B(b, 2)$ are interchanged at the next iteration cycle as shown in Figure 8. In other words, the repetition periods of pointer assignment for arcs with initial samples may span multiple iteration cycles. Section 4 is devoted to computing the repetition period of pointer assignment for the arcs with initial samples.

Suppose an arc a has M local buffers. Since the local buffers are accessed sequentially, each local buffer entry has at most $\lceil \mathrm{TNSE}(a)/M \rceil$ samples and the pointer to sample $s(a, k)$ is stored in $B(a, k \bmod M)$. After the first phase is completed, 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, a code segment is inserted automatically 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.

3.2.2 Static binding strategy

If we use static binding, we may not change the pointer values of local buffers at runtime. It means that all allocated samples to a local buffer should be mapped to the same global buffer. For example of Figure 7, we need six local buffers for static binding: two more buffers than the dynamic binding case since $s(a, 1)$ and $s(a, 5)$ are not mapped to the same global buffer. On the other hand, arc a of Figure 4 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 4 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 8. Note that the head interval of $s_2(b, 3)$ is connected to the tail interval of $s_3(b, 1)$ in the next repetition period. Therefore, four live samples are involved in 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 four 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. We refer to the next section for detailed discussion.

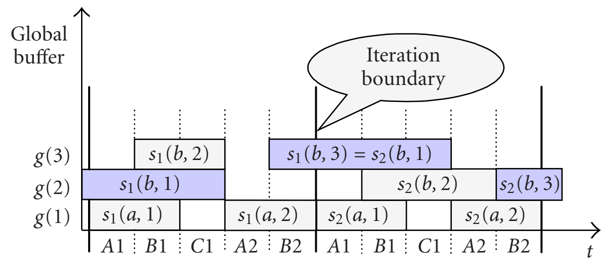


FIGURE 8: The global buffer lifetime chart spanning two iteration cycles for the example of Figure 4.

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. In this section, we show how to compute the repetition period of sample lifetime patterns to determine the periodic pointer assignment for dynamic binding or to determine the minimum size of local buffers for static binding. For simplicity, we assume that all samples have the same size in this section. This assumption will be released in Section 5.

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 $d(a)$ initial samples on an arc a, the lifetime interval of $(d(a) \bmod \mathrm{TNSE}(a))$ newly created samples on the arc spans $\lceil d(a)/\mathrm{TNSE}(a) \rceil + 1$ iteration cycles and that of $(\mathrm{TNSE}(a) - (d(a) \bmod \mathrm{TNSE}(a)))$ samples spans $\lceil d(a)/\mathrm{TNSE}(a) \rceil$ iteration cycles.*

Let p be the number of iteration cycles in which a sample lifetime interval lies. Figure 9 illustrates two patterns that a sample lifetime interval can have in a global lifetime chart. A sample starts its lifetime at the first iteration cycle with a head interval and ends its lifetime at the pth iteration with a tail interval. Note that the tail interval at the pth iteration also appears at the first iteration cycle. The first pattern, as shown in Figure 9a, 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.

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 9b shows a simple case that a new head interval can be placed at the next iteration cycle. Then, the repetition period of the sample

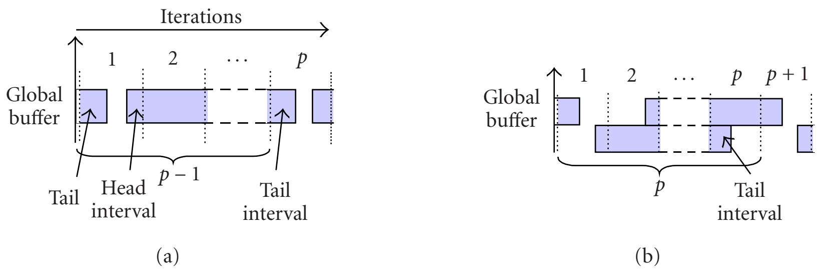
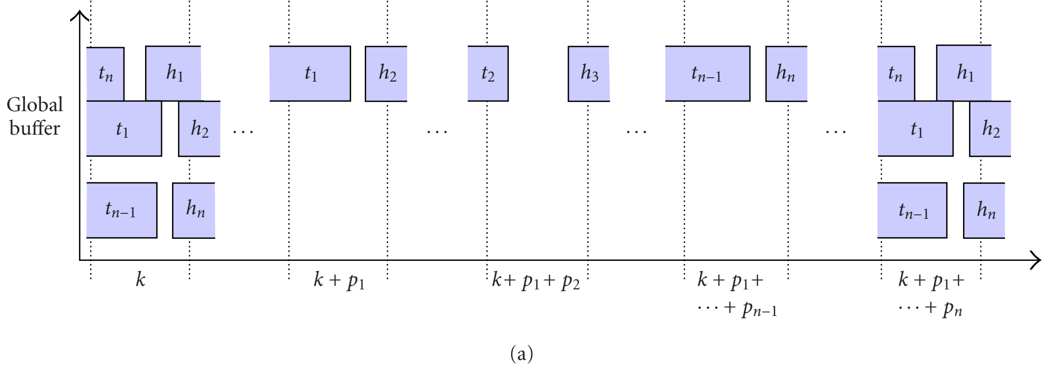


FIGURE 9: Illustration of a sample lifetime interval: (a) when the tail interval is mapped to the same global buffer as the head interval, and (b) when the tail interval is mapped to a different global buffer and there is no chained multicycle sample lifetime interval.



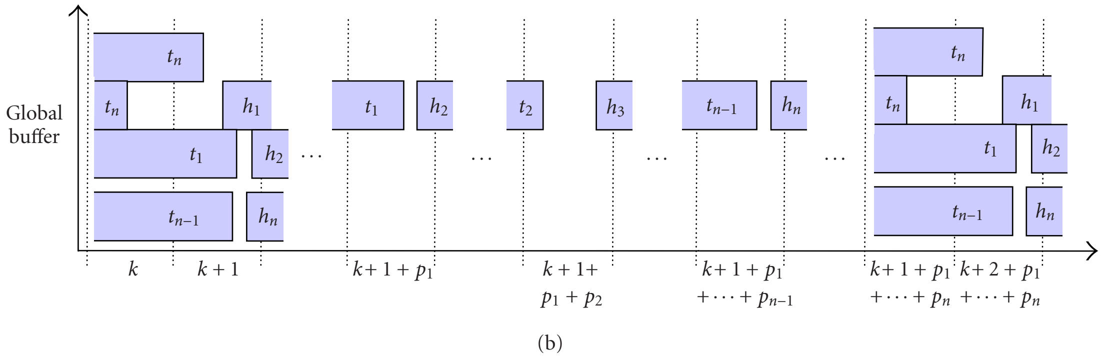
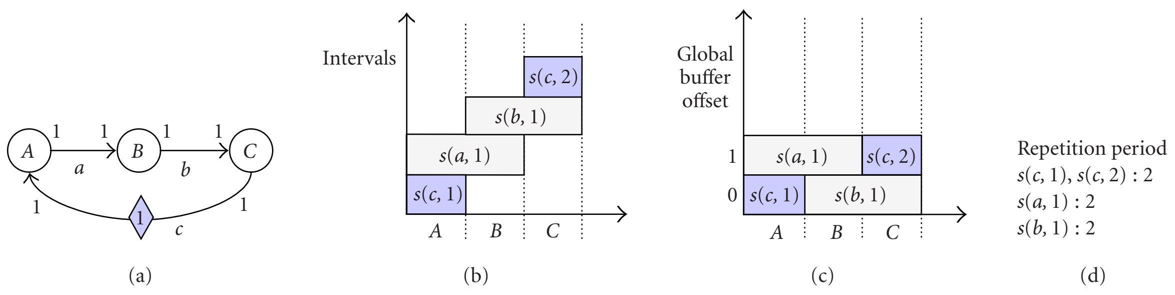


FIGURE 10: Sample lifetime patterns when multicycle lifetimes are chained so that tail interval t_i is chained to the lifetime of sample $j + 1$. (a) Case 1: t_n is chained back to the lifetime of sample 1. The repetition period of sample lifetime patterns becomes $\sum_{i=1}^{n} p_i$. (b) Case 2: t_n is chained to none. The repetition period becomes $\sum_{i=1}^{n} p_i + 1$. Here, we assume that the lifetime of sample k spans $p_k + 1$ iteration cycles.

lifetime pattern becomes p. More general case occurs when another multicycle sample lifetime on a different arc is chained after the tail interval. A multicycle 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.

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.*

Proof. Figure 10 illustrates two patterns where chained multicycle lifetime intervals are placed. The horizontal axis indicates the iteration cycles. The lifetime interval of sample 1 starts at k with head interval h_1 and finishes at $k + p_1$ with tail interval t_1. Since the lifetime of sample 2 is chained, its head interval h_2 is placed at the same global buffer as t_1. The lifetime of sample 2 ends $k + p_1 + p_2$. If we repeat this process, we can find that the lifetime of sample n ends at $k + \sum_{i=1}^{n} p_i$. Now, we consider two cases separately. Case 1: when interval t_n is chained back to the lifetime of sample 1, the repetition period becomes $\sum_{i=1}^{n} p_i$ as illustrated in Figure 10a. Case 2: when interval t_n is chained to no more lifetime, we should prove that sample 1 is mapped to the same global buffer at



```
struct frame g[2];
main()
{
        struct G *a, *b, *c[2] = {g, g + 1};
         int in _A = 0, out _C = 1;
        for(int i = 0; i < max _iteration; i++) {
                {       a = c[(i + 1)%2];
                        // A's codes. Use c[in _A] and a.
                        in _A = (in _A + 1)%2;
                }
                {       b = c[i %2];
                        // B's codes. Use a and b.
                }
                {       // C's codes. Use b and c[out _C].
                        out _C = (out _C + 1)%2;
                }
        }
}
```

(e)

```
struct frame g[2];
main()
{
    struct G *a[2] = {g + 1, g}, *b[2] = {g, g + 1}, *c[2] = {g, g + 1};
    int in _A = 0, out _A = 0, in _B = 0, out _B = 0, in _C = 0, out _C = 1;
    for(int i = 0; i < max _iteration; i++) {
            {   // A's codes. Use c[in _A] and a[out _A].
                in _A = (in _A + 1)%2; out _A = (out _A + 1)%2;
            }
            {   // B's codes. Use a[in _B] and b[out _B].
                in _B = (in _B + 1)%2; out _B = (out _B + 1)%2;
            }
            {   // C's codes. Use b[in _C] and c[out _C].
                in _C = (in _C + 1)%2; out _C = (out _C + 1)%2;
            }
    }
}
```

(f)

Figure 11: (a) A graph which is equivalent to Figure 3a, (b) lifetime intervals of samples for an iteration cycle, (c) an optimal global buffer lifetime chart, (d) repetition periods of sample lifetime patterns, (e) generated code with dynamic binding, and (f) generated code with static binding.

the next iteration cycle as shown in Figure 10b. Then, the period becomes $\sum_{i=1}^{n} p_i + 1$. Since the sample lifetime patterns over iteration cycles are permutations of each other, sample 1 should be mapped to among n global buffers assigned to samples 1 through n during previous iterations. As illustrated in Figure 10b, other global buffers are occupied by other samples at $k + \sum_{i=1}^{n} p_i + 1$ except the global buffer mapped to t_n. Therefore, sample 1 is mapped to the same global buffer at the next iteration cycle. □

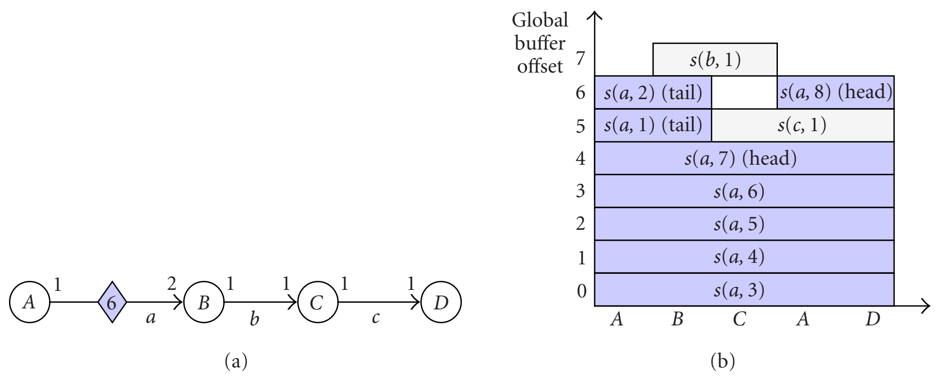
We apply the above theorem to the case of Figure 4b where head interval $s(b, 3)$ and tail interval $s(b, 1)$ are mapped to the different global buffers. And the sample lifetime spans two iteration cycles. Therefore, the repetition period becomes 2 and Figure 8 confirms it.

Another example graph is shown in Figure 11a, which is identical to the simplified H.263 encoder example of Figure 3. 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 11b. If we expand the lifetime chart over two iteration cycles, we can notice that head interval $s_1(c, 2)$ is extended to tail interval $s_2(c, 1)$ at the second iteration cycle. By interval scheduling, an optimal mapping is found like Figure 11c. 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. In Figure 11c, 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 11d.

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. Figures 11e and 11f show the code segments that highlight the difference.

The dynamic binding scheme allocates a local pointer buffer onto arc AB since the number of samples accumulated on arc AB is no greater than one. Similarly, a local buffer is allocated on arc BC. Figure 11e shows a code with dynamic



Repetition period
$s(a, 1), s(a, 3), s(a, 5), s(a, 7) : 4$
$s(a, 2), s(a, 4), s(a, 6), s(a, 8) : 3$
$s(b, 1) : 1$
$s(c, 1) : 4$

(c)

```
struct Gg[8];
main()
{
        struct G *a0[4] = {g + 4, g, g + 2, g + 5}, *a1[3] = {g + 6, g + 1, g + 3},
                 *b[1] = {g + 7}, *c[1] = {0};

        for(int i = 0; i < max_iteration; i++) {
                  {          struct G *output = a0[(i + 3)%4];
                             // A's codes
                  }{         struct G *input[2];
                             input[0] = a0[i%4]; input[1] = a1[i%3];
                             // B's codes
                  }{         c[0] = a0[i%4];
                             // C's codes
                  }{         struct G *output = a1[(i + 2)%3];
                             // A's codes
                  }{
                             // D's codes
                  }
        }
}
```

(d)

Figure 12: (a) An SDF graph with large initial samples, (b) an optimal global buffer lifetime chart, (c) repetition periods of sample lifetime patterns, and (d) generated code with dynamic binding after dividing local buffers on arc AB into two local buffer arrays.

binding. When the size of a local buffer is the same as the number of newly produced samples within an iteration, no buffer index is needed for the buffer in the generated inlined code. The mapped offset of sample $s(a, 1)$ repeats every other cycles as that of $s(c, 2)$ does. The mapped offset of $s(b, 1)$ follows that of $s(c, 1)$. For arc CA, the minimum size of local buffers is one since there is at most a live sample on the arc. But we notice that if we have a local buffer on the arc, we need to update the pointer value of each local buffer at every access since the repetition period is two. Therefore, we allocate two local buffers on arc CA and fix the buffer pointers. Instead, we update the local buffer indices, in_A for block A and out_C for block C. The decision of the binding scheme is automatically taken care of by the algorithm.

The static binding requires two local pointer buffers for arc AB and BC, respectively, since the mapping patterns of samples on AB repeat every other iteration cycles. The local buffer size for arc CA is two and has the same binding as Figure 11e. Figure 11f represents a generated code with static binding, which additionally requires buffer indices for local buffers on arc AB and BC [16]. Hence, we add additional code of updating buffer indices before and after the associated block's execution. We should consider this overhead to compare the static binding with the dynamic binding strategies. In this example, using the dynamic binding strategy is more advantageous.

We illustrate an example graph which has large initial delays and thus has long repetition period of sample lifetime patterns in Figure 12. The schedule is assumed to be given as $ABCAD$. Interestingly enough, samples on the same arc AB have different repetition periods. The mapping patterns repeat every four iteration cycles for samples $s(a, 1)$, $s(a, 3)$, $s(a, 5)$, and $s(a, 7)$ since each sample spans four iteration cycles and tail interval $s(a, 1)$ is not mapped to the same global

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

Figure 13: Pseudocode of LOES algorithm.

buffer as head interval $s(a, 7)$. On the other hand, samples $s(a, 2)$, $s(a, 4)$, $s(a, 6)$, and $s(a, 8)$ repeat their lifetime patterns every three iteration cycles since tail interval $s(a, 2)$ and head interval $s(a, 8)$ are mapped to the same global buffer. The static binding method allocates twelve local buffers to arc AB since the overall repetition period of local buffers on arc AB becomes twelve that is equal to the least common multiple of 4 and 3 ($=$ LCM(4, 3)). The dynamic binding method, however, allots two local buffer arrays that have four and three buffers, respectively, to arc AB. Hence the dynamic binding method can reduce five local pointer buffers than the static binding. A code template with inlined coding style is displayed in Figure 12d. The local buffer pointer for arc CD follows that of sample $s(a, 1)$.

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

5. BUFFER SHARING FOR DIFFERENT SIZE SAMPLES WITHOUT DELAYS

We are given sample lifetime intervals which are determined from 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 NP-hard. 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. We propose a simple but efficient heuristic algorithm. If the graph has no delays (initial samples), we can repeat the assignment every iteration cycle. Graphs with initial samples will be discussed in the next section.

The proposed heuristic is called LOES (lowest offset 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 interval is 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 13. In this pseudocode, 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.

Since the LOES algorithm can find intervals with lowest offset in $O(n)$ time and choose the earliest interval among them in $O(n)$, where n is the number of lifetime intervals, it has $O(n)$ time complexity to assign an interval. Therefore the time complexity of the algorithm is $O(n^2)$ for n intervals.

Figure 14 shows an example graph where the circled number on each arc indicates the sample size. Figure 14b presents a schedule result and the resultant sample lifetime intervals. Figure 15 shows the procedure of the LOES algorithm at work. At first, we select d with the earliest start time first among the intervals that can be mapped to lowest offset 0. Next, f is selected and placed since it is the only interval that can be placed at offset 0. In this example, the LOES algorithm produces an optimal assignment result. With randomly generated graphs, it gives near-optimal results most of the time as shown later.

De Greef et al. proposed a similar heuristic that considers the offset first and sample size next in [12]. Even though

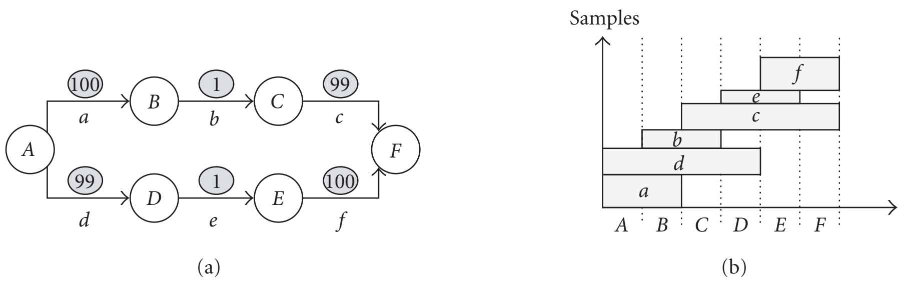


FIGURE 14: (a) An input graph with samples of different sizes and (b) a schedule (= *ABCDEF*) and the resultant sample lifetime chart.

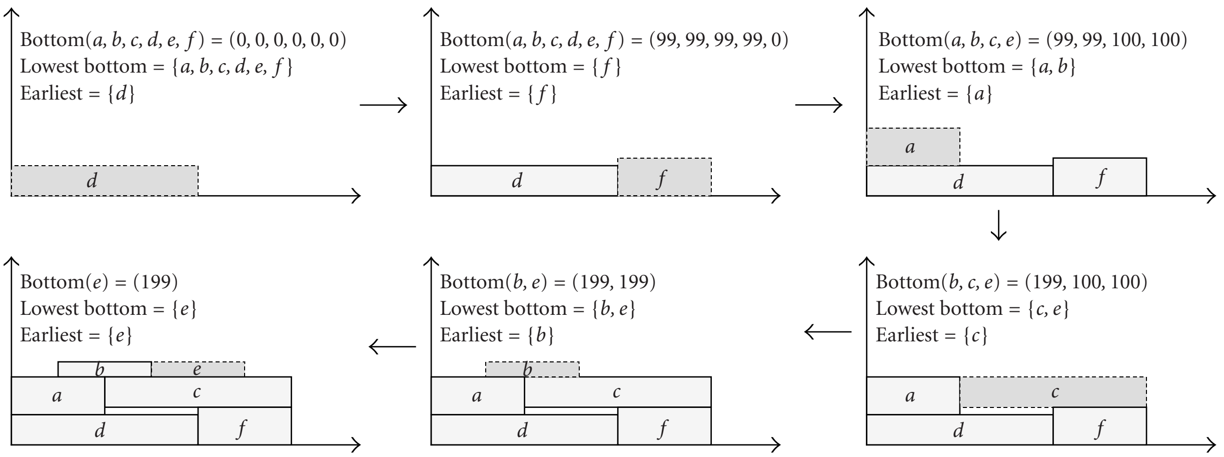

Bottom(a, b, c, d, e, f) = (0, 0, 0, 0, 0, 0)
Lowest bottom = {a, b, c, d, e, f}
Earliest = {d}
Bottom(a, b, c, d, e, f) = (99, 99, 99, 99, 0)
Lowest bottom = {f}
Earliest = {f}
Bottom(a, b, c, e) = (99, 99, 100, 100)
Lowest bottom = {a, b}
Earliest = {a}
Bottom(b, c, e) = (199, 100, 100)
Lowest bottom = {c, e}
Earliest = {c}
Bottom(b, e) = (199, 199)
Lowest bottom = {b, e}
Earliest = {b}
Bottom(e) = (199)
Lowest bottom = {e}
Earliest = {e}

FIGURE 15: The proposed placement algorithm at work.

their heuristic gives similar performance with randomly generated graphs, it does not guarantee to produce optimal results when all samples have the same size.

6. BUFFER SHARING FOR DIFFERENT SAMPLE SIZES WITH INITIAL SAMPLE DELAYS

In this section, we discuss the most general case where a graph has initial samples and samples have different sizes. The LOES algorithm is not directly applicable to this case. Figure 16 illustrates the difficulty with a simple example. Figure 16a shows a mapping result after the LOES algorithm is applied to the first iteration period. 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 16b 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, we enforce that multicycle intervals do not share the global buffer space with other intervals with different sample size.

Figure 17 displays the main procedure, ASSIGN_MAIN, for the proposed technique. We first classify intervals into several groups (lines 2–13 in Figure 17); a new group is formed with all intervals of the same size if there is at least one multicycle interval, and is denoted as $D(x)$ where x is the sample size. If there is no multicycle interval, all remaining intervals form the last group R. Consider an example of Figure 18a where sample sizes are indicated as circled numbers on the arcs. The sample lifetimes are displayed in Figure 18b. We make three groups of intervals for this graph. Group $D(100)$ includes all sample intervals associated with arcs b, c, and d while group $D(200)$ includes all intervals associated with arcs a and e. Initially group R is empty in this example.

The next step is to apply the LOES algorithm for each group $D(x)$ since $D(x)$ contains samples of the same size only (lines 14–17). We slightly modify the LOES algorithm so that the algorithm finishes the mapping as soon as all multicycle intervals are mapped: compare line 24 of Figure 17 and line 12 of Figure 13. The remaining unmapped intervals are moved to group R. In Figure 18c, the modified LOES algorithm places intervals in $D(100)$ in the order of $[s(c,1), s(b,2), s(d,2), s(d,1), s(b,1), s(c,3)]$. After placing $s(c,3)$, it completes and moves a remaining interval $s(c,2)$

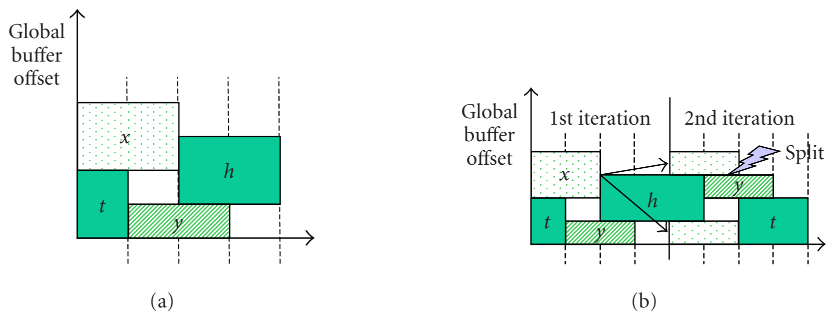


Figure 16: (a) A global buffer lifetime chart with different size samples where t is a tail interval and h is a head interval. (b) Interval x should be split at the second iteration, which is not allowed in the generated code.

```
1: Procedure ASSIGN_MAIN($U$ is a set of sample lifetimes) {
2:     $R \leftarrow \{\}$
3:     for each $x$ in $U${/* classify intervals and sort them */
4:         if ($x$ is delayed interval)
5:             $D(\text{size}(x)) \leftarrow D(\text{size}(x)) \bigcup \{x\}$
6:         else    $R \leftarrow R \bigcup \{x\}$
7:     }
8:     for each $x$ in $R${/* add ordinary intervals into $D(x)$ */
9:         if(there is a delayed interval whose size is equal to that of $x$) {
10:            $D(\text{size}(x)) \leftarrow D(\text{size}(x)) \bigcup \{x\}$
11:            $R \leftarrow R - \{x\}$
12:        }
13:    }
14:    for each $D(\text{size})${ /* place intervals in $D(x)$ */
15:        call M_LOES ($D(\text{size})$).
16:        $R \leftarrow R \bigcup D(\text{size})$
17:    }
18:    /* place ordinary intervals in $R$ */
19:    call LOES($R$).
20:    }
21:}
22: Procedure M_LOES($U$ is a set of sample lifetimes) {     /* slightly modified LOES */
23:    $P \leftarrow \{\}$
24:    While($U$ has delayed intervals) {
           . . .
```

Figure 17: Pseudocode of the proposed algorithm for a graph with delays.

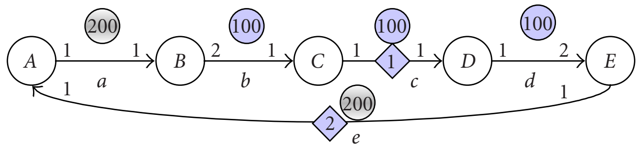
into R. Similarly, ordinary interval $s(e, 2)$ and $s(a, 1)$ are moved into R after $s(e, 3)$ is placed for group $D(200)$. At last we apply the original LOES algorithm to group R (line 19 of Figure 17) as shown in Figure 18e. The algorithm locates intervals $s(c, 2)$, $s(e, 2)$, and $s(a, 1)$ in R as shown in Figure 18e.

After all intervals are mapped to the global buffers, we move to the next stage of determining the local buffer sizes, which is already discussed in Section 4. Repetition periods for $s(c, 1)$ and $s(c, 3)$ become two since tail interval $s(c, 1)$ spans two iteration cycles and is not mapped to the same global buffer as $s(c, 3)$ is. Repetition periods of $s(e, 1)$, $s(e, 2)$, and $s(e, 3)$ become all two. A generated code with static binding is displayed in Figure 18f.

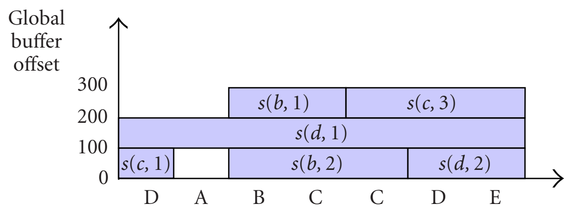
7. EXPERIMENT

In this section, we demonstrate the performance of the proposed technique with three real-life examples and randomly generated graphs.

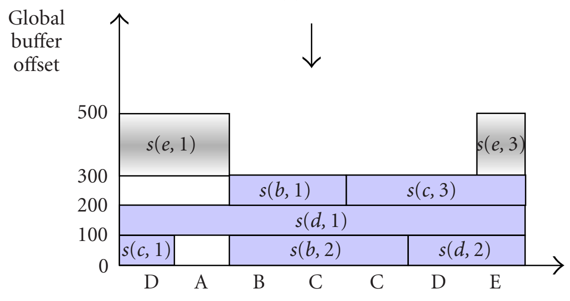
First, we experimented three real-life applications: a JPEG encoder, an MP3 decoder, and an H.263 encoder. We have implemented the proposed technique in a design environment called PeaCE Ptolemy extension as Codesign Environment (http://peace.snu.ac.kr/research/peace) in which dataflow program graphs for a JPEG encoder and an H.263 encoder are displayed in Figures 19 and 20, respectively. From the automatically synthesized codes, we compute the buffer memory requirements and summarize the



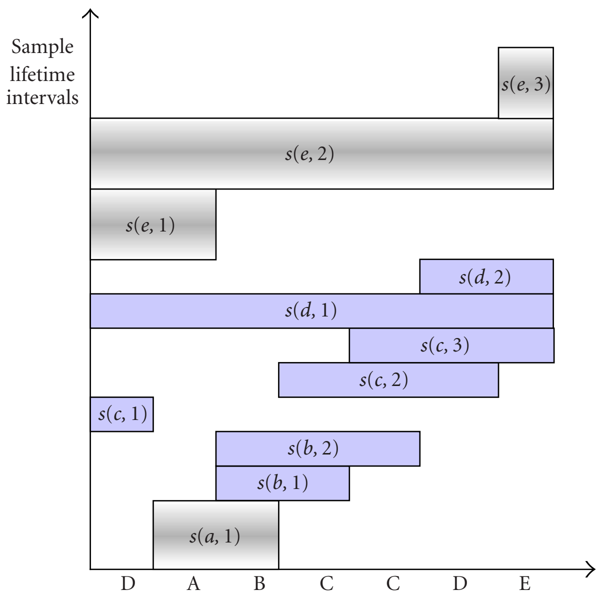
(a)



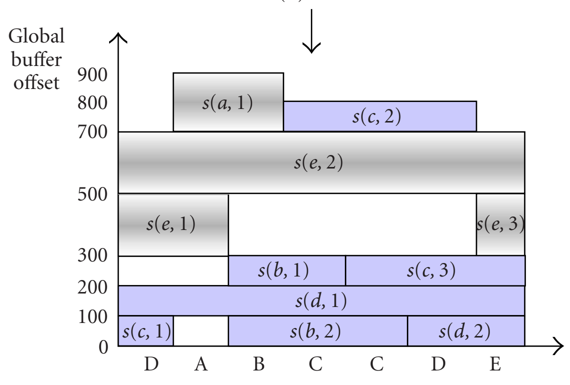
(c)



(d)



(b)



(e)

```
char g[900];
main()
{
          struct G100 *b[4] = {g + 200, g, g, g + 200},
                      * c[4] = {g, g + 700, g + 200, g + 700},
                      * d[4] = {g + 100, g, g + 100, g + 200};
          struct G200 *a[1] = {g + 700}, *e[2] = {g + 300, g + 500};

          for(int i = 0; i < max_iteration; i + +) {
                      {          // D's codes
                      }{         // A's codes
                      }{         // B's codes
                      }{         // C's codes
                      }{         // C's codes
                      }{         // D's codes
                      }{         // E's codes
                      }
          }
}
```

(f)

Figure 18: (a) A graph with samples of different sizes and delays, (b) a sample lifetime chart, (c) LOES placement of samples whose size is 100, (d) LOES placement of samples whose size is 200, (e) LOES placement of the remained samples without delay, and (f) a generated code with static binding.

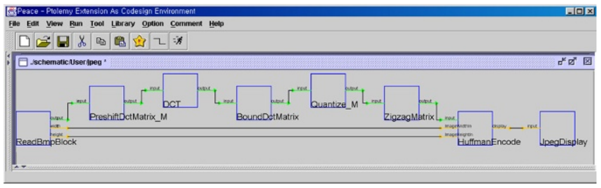


FIGURE 19: JPEG encoder example that represents a graph of same size samples without delays.

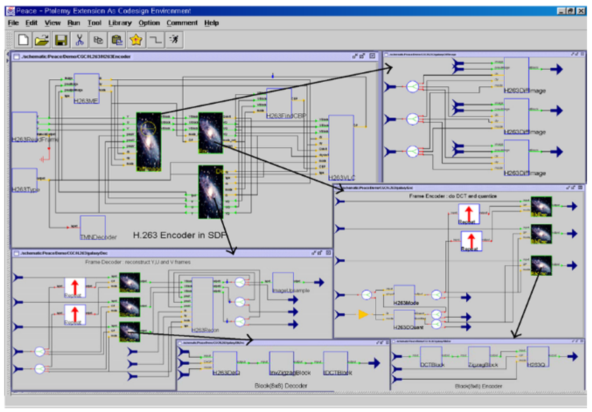

Peace - Ptolemy Extension As Codesign Environment
File Edit View Run Tool Library Option Comment Help
H.263 Encoder in SDF
Frame Encoder : do DCT and quantize
Frame Decoder : reconstruct Y,U and V frames
Block(8x8) Decoder
Block(8x8) Encoder

FIGURE 20: H.263 encoder example that represents a graph of different size samples with delays.

performance comparison results in Table 1. Since the function body of each dataflow node is equivalent in all experiments, only buffer memory requirements are the main item of comparison and the execution times are all similar except for the buffer copy overhead to be discussed below.

A JPEG encoder example represents the first and the simplest class of program graphs in which all nonprimitive data samples have the same size and no initial delay exists. Since the local buffer size of each arc is one in this example, we do not have to separate the local pointer buffer and the global data buffer, which is taken into account in the implementation of the proposed technique. We can reduce the memory requirements to one third as the third and the fourth rows of Table 1 illustrate. The last row indicates the lower bound of global buffer requirements that a given execution sequence of nodes needs. The lower bound is nothing but the maximum total size of data samples live at any instance of time during an execution period. No better result is possible since it is optimal.

An MP3 decoder 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. In this example, we also do not have to separate local and global buffers because the local buffer size of each arc is one. 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. The proposed algorithm also achieves the lower bound in this example.

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 nonprimitive 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 four different size sample sizes and eight 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 neither buffer separation nor sharing between different samples is considered. In this sample, we cannot achieve the lower bound but 256 bytes larger buffer space. Note that the lower bound is usually not achievable if different size samples and initial samples are involved in the same graph.

The SDF model has a limitation that it regards a sample of nonprimitive 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 macroblocks, paying nonnegligible data copy overhead and extra frame-size buffer space. In a manually written reference code, such data copy is replaced with pointer operation. Table 2 reveals this observation: that even the lower bound of memory requirements of the synthesized code from the SDF model is greater than that of the reference code. Therefore, we apply the proposed technique to an extended SDF model, called cyclo-static dataflow (CSDF) [17]. With the CSDF model, we could remove such data copy overhead. And the proposed buffer sharing technique further reduce the memory requirement by 17% more than the reference code.

In the experiments, we choose the better binding strategy, static or dynamic, for each data samples, considering the buffer memory and the code overhead of index updates. In the H.263 encoder example, static binding is preferred for places where the repetition periods of sample lifetimes span more than one iteration cycle. In this example, the pointer referencing through local buffers incurs runtime overhead, which is about 0.16% compared with the total execution time in the H.263 encoder.

The second set of experiments as shown in Table 3 have been performed to evaluate the proposed LOES heuristic. We compare it with an integer linear programming (ILP) solver, CPLEX (http://www.cplex.com). We randomly generate the sample lifetimes within a first iteration interval, varying the start/end times, sizes, and initial delays. When the number of sample intervals exceeds 20, the ILP solver takes prohibitively long execution times. With small size problems below 20 sample intervals, the overhead of LOES heuristic is less than 1% on the average for 150 experiments.

Table 1: Comparison of buffer memory requirements for three real-life examples.

Example	JPEG	MP3	Simplified H.263 encoder	H.263 encoder
Class	Same size No delay	Different size No delay	Same size With delay	Different size With delay
# of samples	6	336	3	1804
Without buffer sharing	1536 B	36 KB	111 KB	659 KB
Sharing buffers of same size only	512 B	23 KB	111 KB	510 KB
Buffer sharing without buffer separation	512 B	11 KB	111 KB	510 KB
Buffer sharing with buffer separation	—	—	74 KB	396 KB
Lower bound of global buffer size	512 B	11 KB	74 KB	396 KB

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

Example	Reference code	Buffer sharing without buffer separation in CSDF	Buffer sharing with buffer separation in CSDF
H.263 Encoder	350 KB	291 KB	290 KB

Table 3: Performance comparison of LOES algorithm with integer linear programming (optimal) for randomly generated graphs (unit: %).

# of intervals	5	7	9	11	13	15	20	Avg.	max
(LOES-ILP)/ILP	0.0	0.0	0.1	0.1	0.5	0.3	0.3	0.2	14

8. CONCLUSIONS

We have proposed a buffer sharing technique for data samples of nonprimitive type to minimize the buffer memory requirements from graphical dataflow programs based on the SDF model or its extension assuming that the execution order of nodes is already determined at compile time. In order to share more buffers, the proposed technique separates global memory buffers from local pointer buffers: the global buffers store live samples and the local buffers store the pointers to the global buffer entries. The technique minimizes the buffer memory by sharing global buffers for data samples of different size. No previous work is known to us to solve this sharing problem especially for the graphs with initial samples. It also involves three subproblems of mapping local pointer buffers onto the global buffer space, determining the local buffer sizes, and finding the repeating mapping patterns.

We first obtain the minimum size of global buffer spaces assuming that local pointer buffers take negligible amount of buffer space compared with the global buffer space. A LOES algorithm has been developed for buffer sharing between samples of different sizes. The next step was to bind the local pointer buffers to the given global buffers for the graph. We present both dynamic binding and static binding methods and compare them in terms of memory requirements and code overheads. The proposed technique, including automatic code generation and memory optimization, has been implemented in a block diagram design environment called PeaCE. No manual intervention is necessary for the proposed code generation technique in PeaCE.

The experimental results show that the proposed algorithm is useful, especially for the graphs with initial delays. The proposed algorithm that separates local buffers and global buffers reduce more memory by 33% in the simplified H.263 encoder and 22% in the H.263 encoder than the sharing algorithm that does not separate local buffers and global buffers. Through extensive buffer sharing optimizations, automatic software synthesis from a dataflow program graph achieves the comparable code quality with the manually optimized code in terms of memory requirement.

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 algorithm.

ACKNOWLEDGMENTS

This research is supported by National Research Laboratory Program (no. M1-0104-00-0015) and Brain Korea 21 Program. The RIACT at Seoul National University provides research facilities for this study.

REFERENCES

[1] *COSSAP User's Manual, Synopsys*, Mountain View, Calif, USA.

[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, pp. 32–43, 1990.

[3] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: a framework for simulating and prototyping heterogeneous systems," *International Journal of Computer Simulations*, vol. 4, no. 2, pp. 155–182, 1994.

[4] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous dataflow programs for digital signal processing," *IEEE Trans. on Computers*, vol. 36, no. 1, pp. 24–35, 1987.

[5] Telenor Research, "TMN (H.263) Encoder/Decoder Version 2.0," June 1997, ftp://bonde.nta.no/.

[6] P. K. Murthy, S. S. Bhattacharyya, and E. A. Lee, "Joint minimization of code and data for synchronous dataflow programs," *Journal of Formal Methods in System Design*, vol. 11, no. 1, pp. 41–70, 1997.

[7] W. Sung, J. Kim, and S. Ha, "Memory efficient software synthesis from dataflow graph," in *11th International Symposium on System Synthesis (ISSS '98)*, pp. 137–144, Hsinchu, Taiwan, December 1998.

[8] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations," *Journal of Design Automation for Embedded Systems*, vol. 2, no. 1, pp. 33–60, 1997.

[9] F. J. Kurdahi and A. C. Parker, "REAL: a program for register allocation," in *Proc. 24th ACM/IEEE Design Automation Conference (DAC '87)*, pp. 210–215, Miami Beach, Fla, USA, June 1987.

[10] M. R. Garey, D. S. Johnson, and L. J. Stockmeyer, "Some simplified NP-complete graph problems," *Theoretical Computer Science*, vol. 1, no. 3, pp. 237–267, 1976.

[11] J. Gergov, "Algorithms for compile-time memory optimization," in *Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '99)*, pp. 907–908, Baltimore, Md, USA, January 1999.

[12] E. De Greef, F. Catthoor, and H. D. Man, "Array placement for storage size reduction in embedded multimedia systems," in *Proc. International Conference on Application-Specific Array Processors (ASAP)*, pp. 66–75, Zurich, Switzerland, July 1997.

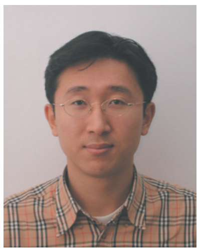
[13] P. K. Murthy and S. S. Bhattacharyya, "Shared buffer implementations of signal processing systems using lifetime analysis techniques," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 2, pp. 177–198, 2001.

[14] S. Ritz, M. Willems, and H. Meyr, "Scheduling for optimum data memory compaction in block diagram oriented software synthesis," in *Proc. IEEE Int. Conf. Acoustics, Speech, Signal Processing*, pp. 2651–2653, Detroit, Mich, USA, May 1995.

[15] F. Gavril, "Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 180–187, 1972.

[16] S. S. Bhattacharyya and E. A. Lee, "Memory management for dataflow programming of multirate signal processing algorithms," *IEEE Trans. Signal Processing*, vol. 42, no. 5, pp. 1190–1201, 1994.

[17] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static data flow," *IEEE Trans. Signal Processing*, vol. 44, no. 2, pp. 397–408, 1996.



Hyunok Oh is a Ph.D. candidate in the School of Computer Science and Engineering at Seoul National University, Korea. He received his B.A. degree (1996) and M.A. degree (1998) in computer engineering from Seoul National University, and he is in a Ph.D. program from 1998 to 2002. He is interested in hardware-software codesign, model of computation, hardware-software cosynthesis, memory optimization, and multimedia applications.

Soonhoi Ha is currently an Associate Professor in the School of Electrical Engineering and Computer Science at Seoul National University. From 1993 to 1994, he worked for Hyundai Electronics Industries Corporation. He received his B.A. degree (1985) and M.A. degree (1987) in electronics engineering from Seoul National University, and Ph.D. (1992) degree in electrical engineering and computer science from University of California, Berkeley. He has worked on Ptolemy project. His research interests include hardware-software codesign, design methodology for embedded systems, and PC clusters. He is a member of the ACM and IEEE Computer Society.