Buffer Memory Optimization for Video Codec Application Modeled in Simulink

Sang-Il Han* **, Xavier Guerin**, Soo-Ik Chae*, Ahmed. A. Jerraya**

*Department of Electrical Engineering,
Seoul National Univ., Seoul, Korea
{sihan,chae}@sdgroup.snu.ac.kr

**SLS Group, TIMA Laboratory
Grenoble, France
{xavier.guerin,ahmed.jerraya}@imag.fr

ABSTRACT

Reduction of the on-chip memory size is a key issue in video codec system design. Because video codec applications involve complex algorithms that are both data-intensive and control-dependent, memory optimization based on global and precise analysis of data and control dependency is required. We generate a memory-efficient C code from a restricted Simulink model, which can represent both data and control dependency explicitly, by applying two buffer memory optimization techniques: copy removal and buffer sharing. Copy removal is performed while parsing the Simulink model. Buffer sharing requires global scheduling and formal lifetime analysis. Experimental results on an H.264 video decoder show that the buffer memory size and execution time of the C code generated by the proposed method are 71% and 32% less than those of the C code produced by Simulink's C code generator, respectively. When compared to the hand written C code, the memory size was reduced by 27% while its execution time was increased by only 3%.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems

General Terms: Algorithms, Experimentation

Keywords: memory size reduction, video codec application, Simulink

1. INTRODUCTION

Current video codec applications require architectures with large memory and high bandwidth. Due to its large power consumption and limited bandwidth, accessing off-chip memory has been considered as a major bottleneck in video codec systems. Since a large on-chip memory increases die size and chip cost, it is important to minimize the demand of on-chip memory, where code and data are stored. The data memory contains constant coefficients as well as the buffers for the inputs and outputs of algorithm blocks. In this paper, we focus on buffer memory optimization to reduce the on-chip memory size.

The video codec applications also require complex high-performance architectures that can process computation-intensive algorithms for higher resolution images. To design such complex architecture within a short design time, it is necessary to explore design space with high-level system specification and to generate SW and HW code automatically. Therefore, minimizing the buffer memory size in automatic code generation from the high-level system specification is becoming one of the key technologies in video codec system design.

Video codec applications include data-intensive algorithms that require multiple large buffers. Furthermore, the control of the algorithms depends on image modes and macroblock modes [1]. To obtain a SW code with a reasonable size of on-chip memory size from the system specification of a video codec application, it is necessary to use a buffer memory optimization method based on a global and precise analysis of data and control dependency.

Most previous work has focused on buffer memory optimization in generating SW code from dataflow specification [2-5]. However, they did not consider control dependency at all because pure dataflow lacks control constructs and the extension with control constructs has the problem of deadlock or buffer overflow [6]. Consequently, they could not be applied directly to the buffer memory minimization problem for video codec applications.

In this paper, we address the problem of minimizing the buffer memory size in automatic code generation for a subset of Simulink, which is able to represent explicitly control-dependent algorithms of video codec applications without deadlock or buffer overflow problem [7].

The main contribution of this work is the implementation of buffer memory optimization and automatic code generation for the system specification that includes explicit conditionals. For buffer memory reduction, we applied two techniques: **copy removal** and **buffer sharing**. The former eliminates redundant buffers and copy operations, which is introduced by explicit controls and delays. It is performed while parsing the Simulink model. The latter exploits disjoint lifetime of buffers. It requires global scheduling and formal lifetime analysis over the entire system specification. We extended an existing scheduling and buffer sharing algorithm for dataflow specification so that it can handle the conditionals in the Simulink models. This paper includes the measurement and analysis on the effects of the implemented techniques for a H.264 video decoder.

The rest of the paper is organized as follows. In Section 2 we describe previous work on automatic SW code generation with reduced buffer memory. We present a target application and the restricted Simulink model for automatic SW code generation in Section 3, and we describe the optimization techniques to reduce buffer memory and memory copy operations in generating C codes in Section 4. We present and analyze several experimental results that show the effect of each optimization technique and

DAC 2006, July 24–28, 2006, San Francisco, California, USA.
Copyright 2006 ACM 1-59593-381-6/06/0007…$5.00.

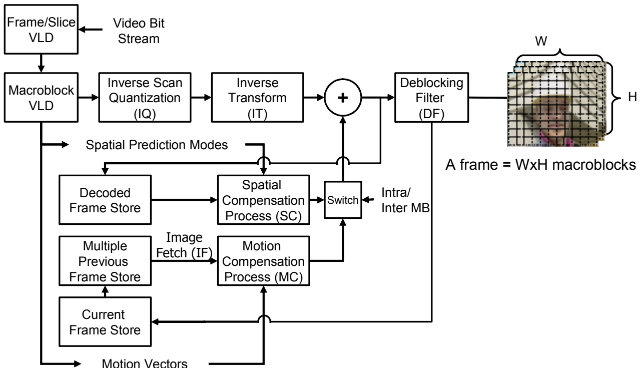


Figure 1. Block diagram of an H.264 decoder.

discuss the limitations of the proposed SW automatic codec generation in Section 5.

2. RELATED WORK

The design approaches based on dataflow specification [6] are widely adopted in designing DSP applications. Ptolemy [8] and COSSAP [9] are well-known dataflow-based design environments and they provide automatic SW generation facilities from dataflow specification. Several previous works proposed buffer sharing [2][3] and scheduling [4][5] techniques for maximizing buffer sharing in SW generation from dataflow specification. However, they didn't address buffer memory minimization for SW code from system specification with explicit control. Note that emerging video codec standards such as H.264 and WMV9 adopt more complex data-dependent operations to improve coding efficiency, which requires a distinct technique to generate a SW code with minimal buffer memory. This paper addresses buffer memory minimization problem when generating a SW code from system specification that includes explicit conditionals.

The data memory minimization problem for sequential programs is a well-known problem [10][11]. Because it is difficult to analyze a large sequential program to extract global data and control dependencies precisely, the existing techniques generally use conservative analysis especially for pointer-intensive programs and/or programs with complex call dependencies. This conservative analysis generally restricts global optimization. We use a system specification that eases global dependency analysis to minimize the memory size for SW code generation.

Pramod et al. addressed a problem of optimizing array storage in MATLAB in [12]. They used an interference graph to represent data dependencies between arrays in a single MATLAB function. They also used a weighted graph coloring to minimize the memory size required to implement the "variables" used in the single MATLAB function. This algorithm shares the same memory space for the arrays only if they have the same intrinsic type. Furthermore, a single MATLAB function may include implicit types that can be resolved with type propagation only through the overall program, so the effect of this optimization algorithm is more limited. In contrast, we resolve implicit type with type propagation and allow overlapping buffers with different intrinsic types.

Real-Time Workshop (RTW) [13] takes a Simulink model [14] as an input and generates a C code as its output. Simulink and RTW have been mainly targeted for control-intensive applications where memory optimization is less important. Consequently, RTW does not provide the memory optimization. We define a subset of Simulink model, which is suitable to modeling video codec applications, for automatic SW generation with buffer optimization.

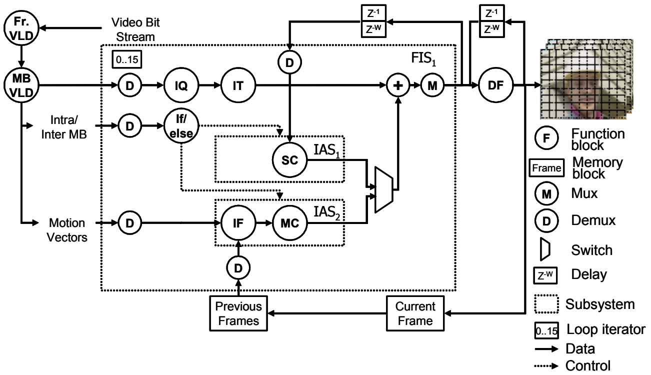


Figure 2. Simplified Simulink model of an H.264 video.

3. THE SUBSET OF SIMULINK FOR VIDEO CODECS

3.1 Video codec applications

Our target applications are macroblock-based video codecs, such as, MPEG-2, H.263, MPEG-4, WMV9 and H.264 [1]. These video codecs are based on hybrid coding, which combines inter-picture prediction to remove temporal redundancy and transformation of the prediction errors to remove spatial redundancy. We will use an H.264 video decoder as a target application example as shown in Figure 1. It receives an encoded video bit stream from a network or a storage device and produces a sequence of frames by applying iterative executions of macroblock-level functions such as variable length decoding (VLD), inverse zigzag and quantization (IQ), inverse transform (IT), spatial compensation (SC), motion compensation (MC) and deblocking filter (DF). The execution paths of these functions are dependent on their image modes, macroblock modes and bitstream contents. For example, macroblock compensation mode ("intra/inter MB" in Figure 1) determines which of spatial compensation and motion compensation is executed.

3.2 The subset of Simulink for video codecs

In order to represent global data and control dependencies precisely, we selected a subset of Simulink for modeling video codec applications. The Simulink subset includes *blocks*, *delays*, *links*, *If-action subsystems* (IAS), and *For-iterator subsystems* (FIS) as well as a global clock that controls the execution of blocks and delays. Figure 2 shows a simplified version of an H.264 video decoder we modeled with the Simulink subset.

- A **block** represents a function. There are two kinds of blocks. One is pre-defined block such as addition, logic AND and so on. The other is user-defined block called "S-function". In Figure 2, "Demux" and "IQ" are pre-defined block and user-defined block, respectively. "Previous Frames" and "Current Frame" are user-defined blocks that represent global memory to store images. Each block has several input and output ports.
- A **delay** Z^{-k} implies that the output is delayed from the input by k clock cycles. In Figure 2, "Z^{-1}" and "Z^{-W}" are examples of delay.
- A **data link** connects one output port of a block or a delay to one or more input ports of one or more blocks and/or delays. A **control link** connects an output of a block to an input port of an If-action subsystem. In Figure 2, the solid line between "IQ" and

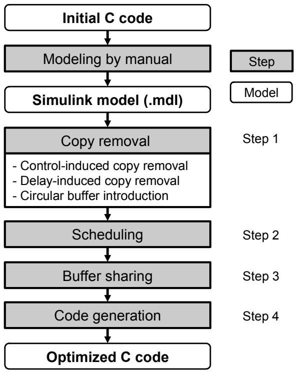


Figure 3. Global procedure of automatic C code

"IT" and the dotted line between "if/else" and "IAS$_1$" are examples of data link and control link respectively.

In order to represent explicit controls, a Simulink model includes two types of subsystems: if-then-else structure and for-loop structure. A subsystem is represented with blocks, delays, links, and other subsystems.

- An **If-action subsystem** (IAS) is used to represent an if-then-else structure. An IAS has a control input port to determine its selection. In Figure 2, "IAS$_1$" and "IAS$_2$" are examples of IAS.
- A **For-iterator subsystem** (FIS) is used to represent a for-loop structure. It is used to describe sequential or parallel repeated executions of blocks where the number of repetitions is known. In Figure 2, "FIS$_1$" are examples of FIS. A "Loop iterator" in a FIS represents the range of loop index (from 0 to 15 in example).

The full H.264 decoder Simulink model consists of 83 user-defined blocks, 101 pre-defined blocks, 286 data links, 43 IASs, 5 FISs, and 24 delays, in which the global clock represents the macroblock index.

We will consider only discrete model with a global clock. We don't handle any other models such as discrete model with multiple clocks and event-driven model since the conventional memory optimization is hard to apply to them.

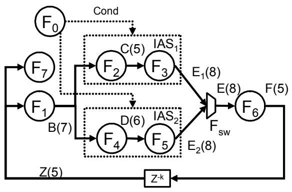
4. AUTOMATIC C CODE GENERATION AND BUFFER MEMORY OPTIMIZATION

4.1 Automatic C code generation

We developed an automatic C code generator, LESCEA (Light and Efficient Simulink Compiler for Embedded Application), which can generate a memory-efficient C code from the restricted Simulink model defined in the previous section. It provides four options for buffer memory optimization, which are control-induced copy removal, delay-induced copy removal, circular buffer introduction, and buffer sharing. Automatic code generation with buffer memory optimization involves the following basic steps as shown in Figure 3.

Step 1. Copy removal: LESCEA first allocates a default buffer memory to each data link according to its data property and then it performs control-induced copy removal, delay-induced copy removal, and circular buffer introduction. This step will be detailed in 4.2.

Step 2. Scheduling: LESCEA determines the invocation order of blocks according to a scheduling policy to maximize buffer sharing. This step will be detailed in 4.3.



(a) Simulink model

```
 1: While(1) {
 2:  F0(cond);
 3:  F1(Z[0],B);
 4:  if(cond) { F2(B,C); F3(C,E1); }
 5:  else     { F4(B,D); F5(D,E2); }
 6:  if(cond) E = E1; else  E = E2;
 7:  F6(E, F);
 8:  F7(Z[0]);
 9:  for(i=0;i<k-1;i++)
10:    Z[i] = Z[i+1];
11:   Z[k-1] = F;
12: }
```

(b) Original C code

```
 1: While(1) {
 2:  F0(cond);
 3:  F1(Z[0],B);
 4:  if(cond) { F2(B,C); F3(C,E); }
 5:  else     { F4(B,D); F5(D,E); }
 6:  F6(E, F);
 7:  F7(Z[0]);
 8:  for(i=0;i<k-1;i++)
 9:    Z[i] = Z[i+1];
10:   Z[k-1] = F;
11: }
```

(c) C code after control induced copy removal

```
 1: While(1) {
 2:  F0(cond);
 3:  F1(Z[0],B);
 4:  if(cond) { F2(B,C); F3(C,E); }
 5:  else     { F4(B,D); F5(D,E); }
 6:  F7(Z[0]);
 7:  F6(E, Z[0]);
 8:  for(i=0;i<k-1;i++)
 9:    Z[i] = Z[i+1];
10:   Z[k-1] = Z[0];
11: }
```

(d) C code after delay induced copy removal

```
1: While(1) {
2:  F0(cond);
3:  F1(Z[clk%k],B);
4:  if(cond) { F2(B,C); F3(C,E); }
5:  else     { F4(B,D); F5(D,E); }
6:  F7(Z[clk%k]);
7:  F6(E, Z[clk%k]);
8: clk++;
9: }
```

(e) C code after circular buffer introducing

Figure 4. Copy removal techniques for a restricted Simulink model.

Step 3. Buffer sharing: LESCEA allows two buffers to share the same memory space if their lifetimes are disjoint. This step will be detailed in 4.4.

Step 4. Code generation: LESCEA generates a code, which includes memory declarations, a sequence of function calls corresponding to the invocation order of blocks, and maps the allocated memory space to the arguments of functions. This step will be detailed presented in 4.5.

4.2 Copy removal

We apply three techniques for copy removal. They are control-induced copy removal, delay-induced copy removal, and circular buffer introduction. These copy removal techniques will be introduced using the example in Figure 4. Figure 4(a) and (b) show a Simulink model and its corresponding original C code, respectively. In Figure 4 (a), each link is annotated with a buffer name and its size. For example, C (5) means buffer C with size 5.

1) **Control-induced copy removal** eliminates copy operations between one or more input buffers and one or more output buffers of multiple I/O Simulink blocks such as "Switch", "Selector", "Mux", and "Demux". These pre-defined Simulink blocks are required to represent explicit conditionals or loops. This technique also allows them to share the same memory space. After applying it to the lines 4 to 6 in Figure 4(b), the input buffers "E$_1$" and "E$_2$'' of the switch are merged with its output buffer "E" in the resulting code in Figure 4(c).

2) **Delay-induced copy removal** eliminates copy operation between the input and output buffers of a delay and allows them to share the same memory space. After applying it to the lines 6, 7, and 10 in Figure 4(c), the line 6, 7 and 10 in the resulting C code in Figure 4(d) is obtained.

3) **Circular buffer introduction** converts a shifting buffer to a circular buffer to implement non-unitary delay. After applying it to the lines 3, and 6 to 10 in Figure 4(d), the lines 3, and 6 to 8 in the resulting code in Figure 4(e) is obtained. This technique introduces a clock variable to index buffer elements, which corresponds to "clk" in Figure 4(e). It does not reduce the memory size, but eliminates copy operations.

These techniques require a scheduling constraint to preserve the specification semantic: all blocks, which read data from a copy inducing block or delay, should be executed before executing the block that writes data to the block or delay. For example,

```
Scheduling(time $t_0$, model M) {
- Step 1. Classify blocks into three sets:
  R : a set of schedulable blocks = {sources and delays in M}
  U : a set of not schedulable blocks = {other blocks in M}.
  S : a set of scheduled blocks S = { }.
  Set time $t$ to $t_0$. Set $s_{liv}(0)$ $s_{max}(0)$ to 0.
- Step 2. Compute the $s_{liv}(t,v)$ and $s_{peak}(t,v)$ for each block $v$ in R.
- Step 3. Select the block $v$ that has the lowest $s_{liv}(t,v)$ and
  satisfies $s_{peak}(t,v) < s_{max}(t)$. If no block is found, select the block
  $v$ that has the lowest $s_{peak}(t,v)$.
- Step 4. Move the select block $v$ from R to S.
    Set the invocation time of the block $v$ to $t$.
    Update $s_{max}(t+1)$, $s_{liv}(t+1)$. Increase time $t$ by one.
- Step 5. Move the schedulable blocks in U to R.
- Step 6. Repeat Step 2-5 until both U and R are empty.
  return <$s_{max}(t)$, $s_{liv}(t)$>}
```

Figure 5. Pseudo code of scheduling algorithm.

executing block F_7 is prior to executing block F_6 in Figure 4(d). LESCEA considers this constraint in the scheduling step.

4.3 Scheduling

LESCEA performs scheduling to find the best possible invocation sequence of blocks to maximize buffer sharing. We extended the existing dataflow based scheduling methods [4][5] for the Simulink subset to support nested conditionals and loops.

Here, each block processing is assumed to be invoked once in a unit of time interval. The lifetime of each buffer is represented as a time interval [*t, t'*) that starts at the invocation time *t* of its source block (included) and ends at the completion time *t'* of the last invoked one among its destination blocks (excluded). The buffer is said to be live during its lifetime, defined at time *t*, and dead at time *t'*. Note that a block should hold both input buffers and output buffers until its completion.

Figure 6 (b) shows an example of the *buffer lifetime chart* [3] that display the lifetimes of buffers. The horizontal axis indicates the abstract time and the vertical axis indicates the memory address offset. Each rectangle denotes the lifetime interval of a buffer. To derive the scheduling algorithm, we need to define some terminologies.

Definition 1. Let $s_{max}(t)$ and let $s_{liv}(t)$ be the maximum of peak memory size and the live memory size after the block, invoked at time *t*-1, completes. Let $s_{def}(t,v)$ and $s_{dead}(t,v)$ be the memory size of defined buffers and that of dead buffers if block *v* is invoked at time *t*, respectively. Similarly, let $s_{peak}(t,v)$ and $s_{liv}(t,v)$ be the peak memory size and the live memory if block *v* is invoked at time *t*. Then they can be defined as follows:

$$s_{liv}(t,v)=s_{liv}(t)+s_{def}(t,v)-s_{dead}(t,v) \quad (1)$$

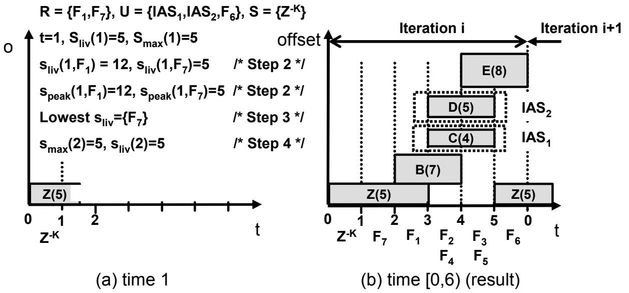
$$s_{peak}(t,v)= s_{liv}(t)+s_{def}(t,v) \quad (2)$$

After block *v*, invoked at time t, completes,

$$s_{liv}(t+1)= s_{liv}(t,v) \quad (3)$$

$$s_{max}(t+1)= \max(s_{max}(t), s_{peak}(t,v)) \quad (4)$$

The objective of scheduling is finding the invocation times of blocks that minimize the maximum of peak memory size over all time. Intuitively, it makes the fat point in buffer lifetime chart as thin as possible. Because this problem is a NP-hard [5], we used a greedy-style algorithm for scheduling. In short, the algorithm selects the block that minimizes the live memory size if it does not increase the maximum of peak memory size. If there is no block that does not increase the maximum of peak memory size, the algorithm selects the block that minimizes the maximum of peak memory size. Figure 5 shows the algorithm pseudo-code that takes a Simulink model as input, and returns the maximum of peak memory size and the live memory size of the model. The algorithm complexity is O(n^2) where n is the number of blocks. In Step 5, a block (and subsystem) is schedulable at time *t* if all of its input buffers are defined before time *t*.


R = {F1,F7}, U = {IAS1,IAS2,F6}, S = {Z-K}
o
t=1, Sliv(1)=5, Smax(1)=5
sliv(1,F1) = 12, sliv(1,F7)=5 /* Step 2 */
speak(1,F1)=12, speak(1,F7)=5 /* Step 2 */
Lowest sliv={F7} /* Step 3 */
smax(2)=5, sliv(2)=5 /* Step 4 */
Z(5)
0 1 2
Z-K
t
offset
Iteration i
Iteration i+1
E(8)
D(5)
IAS2
C(4)
IAS1
B(7)
Z(5)
Z(5)
0 1 2 3 4 5 0
Z-K F7 F1 F2 F3 F6
F4 F5
t

(a) time 1

(b) time [0,6) (result)

Figure 6. A scheduling example.

To compute the peak memory size and the live memory size of a subsystem *v* in step 2, LESCEA applies the algorithm to the subsystem recursively. It uses the following equations instead of equation (1-2) where <$S_{max}(t,v)$, $S_{liv}(t,v)$> = Scheduling(t,v).

$$s_{liv}(t,v)=s_{liv}(t)+ S_{liv}(t,v) \quad (1')$$

$$s_{peak}(t,v)= s_{liv}(t)+ S_{max}(t,v) \quad (2')$$

In the IAS case, LESCEA ignores the increased time and the produced buffers in all the other exclusive IASs.

Figure 6 (a) shows the algorithm steps at time 1 for the Simulink model and its copy removal result as shown in Figure 4 (a) and Figure 4 (e), respectively. Scheduled block set S is {Z^{-K}} and schedulable block set R is {F_1, F_7}. The s_{liv} (1, F_1) and s_{liv} (1, F_7) are 12 and 5, respectively. The s_{peak} (1, F_1) and s_{peak} (1, F_7) are 12 and 5, respectively. The scheduling algorithm selects F_7 because it has the lowest live memory size and does not increase the maximum peak memory size ($S_{max}(1) = 5$).

Figure 6 (b) shows the result of the scheduling algorithm. Note that the invocation times of F_2, F_3 and F_4, F_5 are overlapped because they belong to two exclusive IASs, respectively.

4.4 Buffer sharing

After the scheduling of all the blocks in a Simulink model, LESCEA performs a lifetime-based buffer sharing algorithm.

The objective of buffer sharing is placing buffers at feasible memory address offsets that minimize the required memory. Since buffer sharing problem is NP-complete [3], LESCEA exploits a heuristic algorithm called LOES algorithm in [3] and extends it to consider the conditionals as shown in Figure 7.

Let $m_{offset}(b, P)$ of a unplaced buffer *b* be the feasible address offset given a set of the placed buffer set P. Each unplaced buffer can be overlapped with a placed buffer if they have disjoint lifetime or they belong to two exclusive IASs respectively. In short, the algorithm selects the block that has the lowest offset and the earliest start time [3]. The algorithm complexity is O(n^3) where n is the number of buffers since it computes $m_{offset}(b, P)$ for each *b* in O(n), Step 2 in O(n), and Step 5 in O(n).

Figure 8 shows a buffer sharing example for the Simulink model and its scheduling result shown in Figure 4(a) and Figure 6(b), respectively. In Figure 8 (a), the place buffer set P is {C} and the unplaced buffer set U is {Z,B,D,E}. The feasible address offsets for the members of U are {5,11,5,12}. Among the buffers with the

```
BufferSharing(set of buffers B) {
- Step 1. Classify buffers into two sets:
   U : a set of the unplaced buffers = {all buffers in B}
   P : a set of the placed buffers = { }
- Step 2. Compute the $m_{offset}(b, P)$ for each buffer $b$ in U.
- Step 3. Select the buffer $b$ that has the lowest $m_{offset}(b, P)$
   If more than one buffers have the lowest $m_{offset}$, then the
   buffer that starts no later than others is chosen.
- Step 4. Move the selected buffer from U to P.
   Place the selected buffer $b$ at its $m_{offset}(b, P)$.
- Step 5. Repeat Step 2-4 until U is empty.}
```

Figure 7. Pseudo code of buffer sharing algorithm.

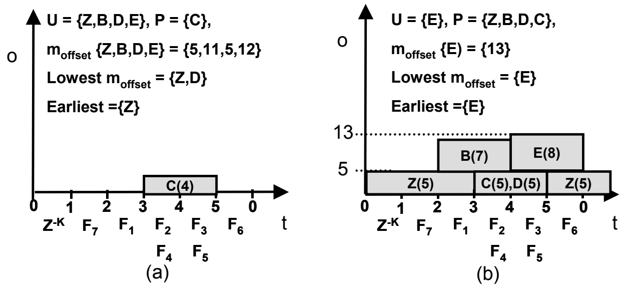


Figure 8. A buffer sharing example.

lowest offset {Z,D}, the earliest buffer is Z. Consequently, LESCEA selects Z and places it at the address 5. Note that the feasible address offset of D is not 9(4(C)+5(D)), but 5(D). The buffers C and D can be shared because each of them belongs to one of the two mutually exclusive IASs, respectively. The circular buffer ("Z" in example), which has a dynamic memory address after circular buffer introduction, can be shared only with the other buffers ("C" and "D" in example) whose sizes are equal to or smaller than its size.

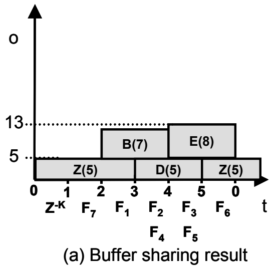
4.5 Code generation

The final step of LESCEA is to generate a code using the buffer sharing result. Figure 9 (a) and (b) show the buffer sharing result and its corresponding code, respectively, for the Simulink model shown in Figure 4(a). The generated code includes 1) memory declarations as lines 1 and 2, 2) a sequence of function calls as lines 4 to 10, 3) control codes as lines 6 and 8, and 4) clock manipulation code as line 11. The functions corresponding to user-defined blocks (F_1-F_7 in example) are called by mapping the allocated memory space to the arguments of the functions. In addition to user-defined blocks, LESCEA can generate C codes corresponding to a large subset of pre-defined blocks such as mathematical operations, logical operations, discrete blocks, etc.

5. EXPERIMENTAL RESULTS

In order to check the effects of the implemented techniques, we generated six versions of C codes from a Simulink model of h.264 baseline decoder with LESCEA by changing the optimization options. We also generated a C code from the Simulink model with RTW. All these versions are described in Table 1 and will be compared to a hand written version of the code.

The target architecture consists of a processor subsystem for image processing and an external global memory for image storage. The processor subsystem consists of an Xtensa processor with a default configuration and an on-chip memory [15]. We profiled the execution cycles of the C codes using Xtensa gdb and gprof when the input bitstream is a sequence of FOREMAN with QCIF format. We traced only the buffers that will be implemented with the on-chip memory.



(a) Buffer sharing result

```
1: int Z[k][5];
2: int mp[8]; // memory pool
3: While(1) {
4:    F7(Z[clk%k]);                      // time 1
5:    F1(Z[clk%k],&mp[0]);               // time 2
6:   if(cond) { F2(&mp[0],Z[clk%k]); // time 3
7:    F3(Z[clk%k],&mp[0]); }             // time 4
8:   else { F4(&mp[0],Z[clk%k]);         // time 3'
9:    F5(Z[clk%k],&mp[0]); }             // time 4'
10:   F6(&mp[0], Z[clk%k]);              // time 5
11:   clk++;
12: }
```

(b) The generated code

Figure 9. An example of the generated code.

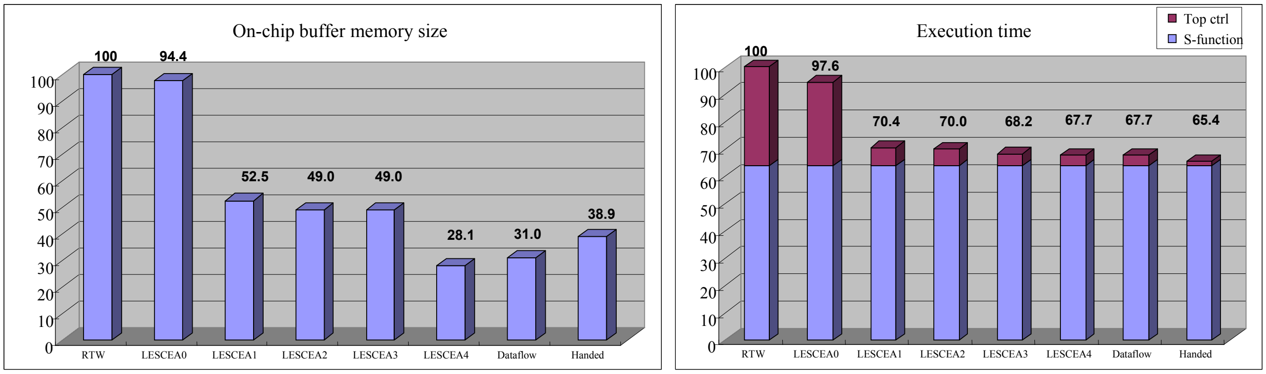
Table 1. C code generation with 8 configurations.

#	Name	Configuration for code generation
1	RTW	RTW.
2	LESCEA0	LESCEA without optimization options.
3	LESCEA1	LESCEA with control-induced copy removal.
4	LESCEA2	LESCEA with control-induced copy removal, and delay-induced copy removal
5	LESCEA3	LESCEA with control-induced copy removal, delay-induced copy removal, and circular buffer introduction
6	LESCEA4	LESCEA with control-induced copy removal, delay-induced copy removal, circular buffer introduction, and buffer sharing.
7	Dataflow	LESCEA with control-induced copy removal, delay-induced copy removal, circular buffer introduction, and buffer sharing (w/o considering conditionals).
8	Handed code	A SW code was optimized manually

Figure 10 shows (a) relative memory size and (b) execution times of the eight configurations. In Figure 10 (a), we found that LESCEA with full optimization options reduces the size of the on-chip buffer memory by 71 % compared to RTW. Note that RTW does not provide any memory minimization techniques, so the buffer memory size of the C code generated with RTW is relatively close to that with LESCEA without optimization option.

The buffer memory size of the generated C code with full optimization options shows even smaller, 27% less, than that of the hand-optimized code. This difference can be explained with two observations. First, to keep readability of the source code, the programmers usually shares two buffer memories only if they are same type and size. Consequently, their buffer sharing is somewhat limited in the hand-optimized code. Second, LESCEA can share the buffer memories more globally across the entire code than the programmer. For example, the Simulink model used in this experiment includes 83 user-defined blocks and 286 data links. It is difficult for the programmer to schedule and allocate such large number of blocks and links respectively while considering global data and control dependency. LESCEA can share the buffer memories with different types and sizes globally even though the readability of its code is worse than the handed written code.

The buffer memory size of the generated C code with full optimization options was 9.4 % less than that of the generated C code without considering conditionals (Dataflow), which can be considered as a trivial extension of dataflow based memory optimization technique.



(a) (b)

Figure 10. Relative memory size and execution times of different configurations.

The buffer memory size of the C code generated with full options is 4364 bytes. The constant memory size is 2172 bytes and the stack memory size is less than 512 bytes.

In Figure 10 (b), "Top ctrl" and "S-function" represent the relative execution time of the generated code and user-defined codes, respectively. The user-defined codes, which are the functions for the video decoder body, are common to all configurations. We found that LESCEA with full optimization options reduces the execution time by 32% compared to RTW because RTW does not eliminate memory copy operations between large-size arrays. Video codec applications include large-sized arrays within nested loops, so we can see that the elimination of the copy operations between the arrays improves significantly the performance of the generated code.

The generated code with LESCEA, however, requires 3% more execution time than the hand written code because LESCEA uses modular instruction, which is not provided by the Xtensa processor with default configuration, to index the circular buffer.

These experimental results show that each buffer memory optimization option affects heavily the memory size and execution time of the generated code from a Simulink model. With these results, we found that system specification that can represent explicitly both data and control dependency is useful for minimization of the buffer memory size in video codec system design.

The current version of LESCEA has several limitations. First, it can take only a subset of Simulink models as its input. For example, it cannot handle a Simulink model with multiple clocks. Second, it does not take into account buffer sharing during the copy removal step. While this approach removes copy operations between large-size arrays, it can affect the result of buffer sharing because copy removal increases the lifetimes of buffer memories. To deal with this problem, we should consider trade-off between copy operations and memory size. Third, the current version of LESCEA can generate only SW code for single-processor systems, which will be extended for multi-processor systems.

6. CONCLUSION

We explained buffer memory optimization techniques and automatic C code generation for a Simulink subset applied for video codec application. We implemented an automatic C code generator called LESCEA and two buffer memory optimization techniques from Simulink. The experimental results for an H.264 video decoder show that the on-chip buffer memory size of the C code generated with LESCEA with full optimization options is reduced by up to 71%, and 27% compared to that with RTW, and manual method, respectively. Additionally the execution time of the code generated with LESCEA is reduced by 32% compared to that with RTW. We found that automatic SW code generation with LESCEA is more effective in minimizing the on-chip memory required for video codec applications than the manual method.

7. REFERENCES

[1] ITU-T and ISO/IEC JTC1, "Advanced video coding for generic audiovisual services," ITU-T Recommendation H.264 – ISO/IEC 14496-10 AVC, 2003.

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

[3] H. Oh and S. Ha, "Memory-optimized Software Synthesis from Dataflow Program Graphs with Large Size Data Samples," EURASIP Journal on Applied Signal Processing Vol. 2003 pp 514-529 May 2003.

[4] S. Ritz, M. Willems, and H. Meyr. "Scheduling for optimum data memory compaction in block diagram oriented software synthesis," Proc. of ICASS'95, May 1995.

[5] Florin Balasa et al, "Background memory area estimation for multidimensional signal processing systems," IEEE TCAD, v.3 n.2, June 1995.

[6] Lee, E. A., Parks, T. M. (1995), "Dataflow process networks," Proc. of the IEEE83(5), pp. 773-801.

[7] S.-I. Han, S.-I. Chae, A.A. Jerraya, "Functional modeling techniques for efficient SW code generation of video codec application" In Proc. of ASP-DAC 2006.

[8] J. T. Buck et al, " Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," International Journal of Computer Simulation, vol. 4, pp. 155-182.

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

[10] J. Fabri, "Automatic storage optimization," in ACM SIGPLAN'79 Notices, 14, #8, pp. 83-91. Aug. 1979.

[11] J. Zhu. "Static memory allocation by pointer analysis and coloring," Proc. of DATE'01, March 2001.

[12] P. G. Joisha, P. Banerjee, "Static array storage optimization in MATLAB," PLDI 2003: pp. 258-268.

[13] Real-Time Workshop, http://www.mathworks.com

[14] Simulink, http://www.mathworks.com

[15] Tensilica Xtensa V, http://www.tensilica.com