Journal of Signal Processing Systems 52, 13–34, 2008
© 2007 Springer Science + Business Media, LLC. Manufactured in The United States.
DOI: 10.1007/s11265-007-0070-9

Optimized RTL Code Generation from Coarse-Grain Dataflow Specification for Fast HW/SW Cosynthesis

HYUNUK JUNG
System LSI Division, CAE Center, Samsung Electronics Co., Ltd., Gyeonggi-Do, South Korea

HOESEOK YANG AND SOONHOI HA
Department of EECS, Seoul National University, Seoul, South Korea

Received: 21 December 2006; Accepted: 20 March 2007

Abstract. This paper presents a new methodology of automatic RTL code generation from coarse-grain dataflow specification for fast HW/SW cosynthesis. A node in a coarse-grain dataflow specification represents a functional block such as FIR and DCT and an arc may deliver multiple data samples per block invocation, which complicates the problem and distinguishes it from behavioral synthesis problem. Given optimized HW library blocks for dataflow nodes, we aim to generate the RTL codes for the entire hardware system including glue logics such as buffer and MUX, and the central controller. In the proposed design methodology, a dataflow graph can be mapped to various hardware structures by changing the resource allocation and schedule information. It simplifies the management of the area/performance tradeoff in hardware design and widens the design space of hardware implementation of a dataflow graph. We also support Fractional Rate Dataflow (FRDF) specification for more efficient hardware implementation. To overcome the additional hardware area overhead in the synthesized architecture, we propose two techniques reducing buffer overhead. Through experiments with some real examples, the usefulness of the proposed technique is demonstrated.

Keywords: HW/SW codesign, system level design, dataflow graph (DFG), RTL, VHDL

1. Introduction

System level design methodology gains considerable research attention as the design complexity and the time-to-market pressure increase for SoC design. In system level design, a system level specification is mapped to an optimal architecture in a systematic way and the mapping is evaluated before implementation for fast design space exploration. As a specification model in this paper, we are concerned with a coarse-grain dataflow model that is adopted in many high level design frameworks [1–4], especially for signal processing and multimedia applications because of formality and readability.

In a dataflow graph G(V,E) as shown in Fig. 1a, an atomic node represents a coarse grain functional block such as FIR filter and DCT, and an arc represents the flow of data samples between two end nodes. When a node is invoked, it consumes the specified number of data samples from each input arc and produces the specified number of samples to each output arc. We use a rather restricted dataflow model, synchronous dataflow (SDF) [5] and its extension to fractional rate dataflow (FRDF) [6], in

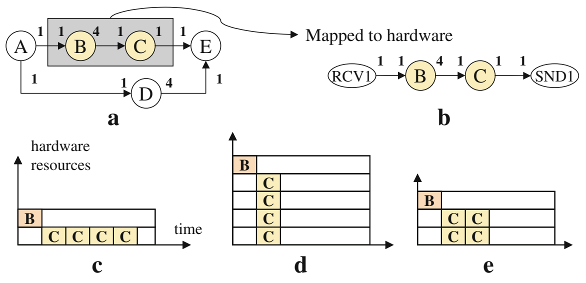


Figure 1. **a** An initial dataflow specification; **b** partitioned subgraph mapped to hardware; **c** fully-sequential, **d** fully-parallel, and **e** hybrid execution of a multi-rate graph.

which the number of data samples produced or consumed on an arc is fixed a priori. If the number, called a sample rate, may not be unity, the dataflow graph is called a multi-rate graph. In Fig. 1, node C can be invoked 4 times after node B is invoked once. This restricted semantics enables us to verify important system properties such as memory boundedness and termination, and to estimate the system performance statically.

A coarse grain block has complex properties such as data sample rates, I/O timings, data types, and its internal states. This dataflow model is different from control/data flow graph (CDFG) [7] commonly used in behavioral synthesis where a node typically represents a basic operation such as add or multiply that can be implemented simply using combinational logic.

We define some terminologies. An invocation of a node is called an *instance* of the node. A hardware implementation of a node is defined as a *hardware resource* associated with the node. A *hardware component* is a physical entity, such as FPGA or ASIC, that integrates all *hardware resources* associated with the mapped nodes.

In the proposed design methodology, a coarse grain functional block is the mapping unit in hardware/software partitioning decision. We assume that functional blocks are library blocks written in C code for software implementation, synthesizable VHDL code for hardware implementation, or both. If a function block is given as a legacy IP block, we may need to add wrapper logic to make it behave as a dataflow node. After hardware/software partitioning is performed, the dataflow graph is partitioned into several graphs that are mapped to hardware or software components. Figure 1b shows an example subgraph mapped to a hardware component augmented with the interface blocks at the subgraph boundary.

Figure 2 shows the simplified HW/SW codesign procedure starting from the dataflow specification. It is assumed that the performance of a node on each processing element (PE, processor or hardware IP) is given in a Node-PE database which also contains the additional information needed for system design such as consumed area and I/O bit width. The initial dataflow specification is first partitioned and scheduled in our design methodology. The partitioning and scheduling is performed by mapping the dataflow nodes to the processing elements in a selected architecture, based on the performance and cost information of nodes. After the partitioning and scheduling is completed for the selected architecture, SW and HW codes are automatically generated and cosimulated to verify the system performance. If it is not satisfied, we go back to the architecture selection step to choose other PEs or architectures. It forms the design space exploration (DSE) loop. In order to accelerate this design loop, it is highly desirable to automate HW and SW code generation from the dataflow specification. This automatic code generation can save time for both coding and debugging.

This paper focuses on the automatic hardware code generation step, which is highlighted in Fig. 2, aiming to accelerate the hardware design and verification. Since the invocation order, or schedule, of dataflow nodes and the required hardware resources are given from the partitioning stage, the hardware code generation problem is to allocate

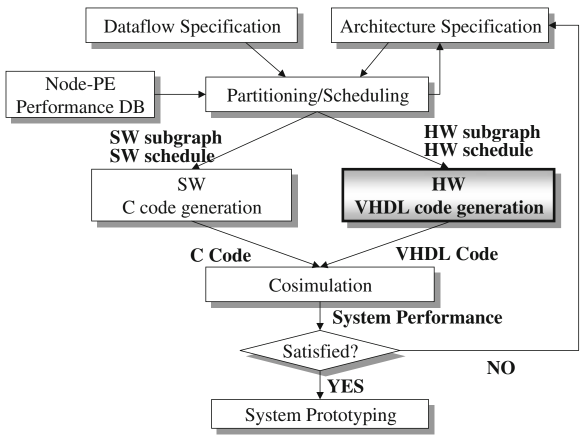


Figure 2. Proposed system design procedure.

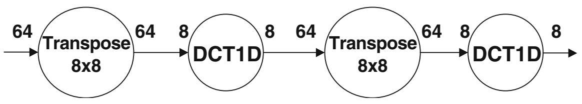


Figure 3. Dataflow specification of 2D DCT algorithm.

hardware resources from the library, and synthesize the interface logic between blocks and the control codes for appropriate clocking and signaling. We enforce the resulting hardware to preserve the dataflow semantics to make the design *correct by construction*.

A multi-rate dataflow graph is very popular in multimedia applications. When synthesizing a hardware component from the multi-rate dataflow graph, we have additional degree of freedom in the design space: parallelism. Even though there have been several works of automatic hardware code generation from dataflow specification, they do not consider the degree of parallelism. There are two different approaches to date: one is fully sequential approach as shown in Fig. 1c [3, 8] where all *instances* of node C are executed sequentially, and the other is fully parallel execution as shown in Fig. 1d [9]. However, the proposed technique explores diverse hardware implementations to consider the performance/area tradeoff: for example a hybrid execution of Fig. 1e is also explored, which is not considered in the previous approaches. The hybrid execution is taken into account in [10] under schedule restriction. But, their focus is different from ours in that they mainly concern the core (SDF node) generation only while we are also considering the efficient control logic generation.

Key contributions in this paper can be summarized as follows:

1. We automate the integration of HW library blocks from an algorithm specification in dataflow model. It enables fast design space exploration by automating the time-consuming and error-prone task of interfacing and integrating HW blocks. We synthesize the glue logics and the central controller to make the HW operate preserving the dataflow semantics according to the schedule information. Therefore, the generated hardware is *correct by construction*.
2. By separating the scheduling and HW code generation, we can implement diverse HW architectures from the given dataflow specification by simply changing the schedule and resource sharing information. It widens the design space of hardware implementation, compared with the previous approaches based on dataflow specification.
3. Since the major overhead which makes the efficiency of generated hardware inferior compared to the manually optimized hardware comes from buffer of SDF arcs, we reduce the buffer size and the control logic dedicated to buffer control by proposing *shift-buffering* and *buffer sharing* schemes. Then, the area of optimized hardware is close to hand-optimized hardware.

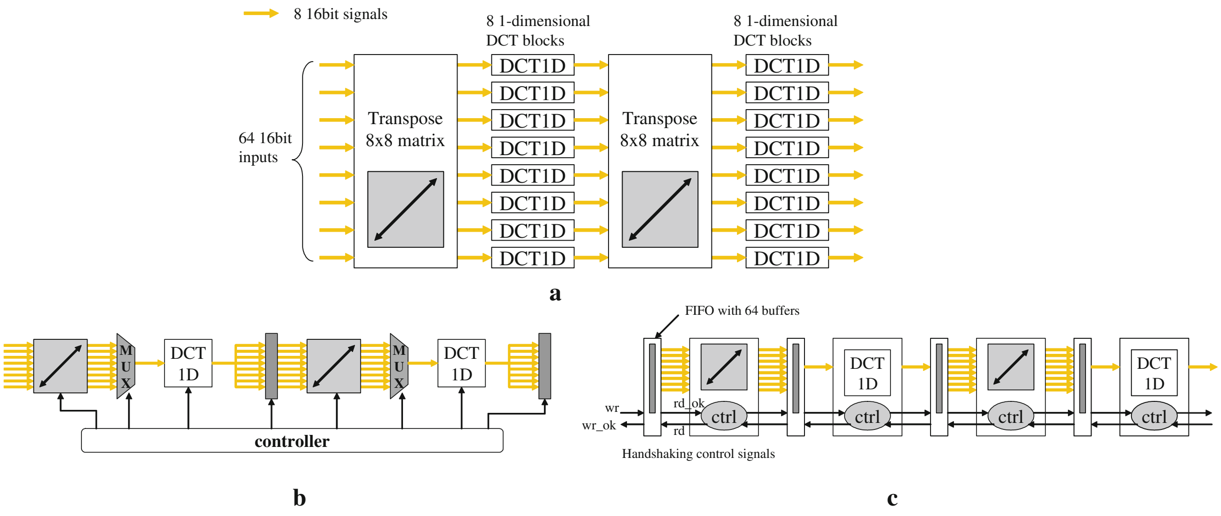


Figure 4. Hardware architecture assumed in **a** Ptolemy0, **b** Meyr's0, and **c** GRAPE0.

In the next section, we overview some related works with a motivational example. After we explain how to define a block in Section 3, we describe the proposed technique in details with examples in Section 4. In Section 5, hardware implementation of a fractional rate dataflow specification is explained for more efficient implementation. Two buffer reducing techniques are proposed in Section 6, followed by experimental results in Section 7. Section 8 concludes the paper with discussion on the remaining topics in this subject.

2. Previous Work and Motivational Example

In order to achieve fast HW/SW cosynthesis, many works have been done to automate hardware implementation from high-level specification in an HDL(Hardware Description Language) or from software specification in C/C++. Behavioral level synthesis from HDLs has a rather long history but with only a limited success. Recently HW synthesis from C or C++ has been actively pursued [11–13] in the realm of ESL (electronic system level) design. While they mainly concern about implementation of a hardware block itself, our approach focuses on implementing the hardware structure in the system level using the predefined library blocks.

Hardware synthesis from SDL specification [14, 15] has been developed for rapid prototyping. This approach is similar to ours in that it is developed for system level design. However, its specification model and application domain are different from ours. It uses asynchronously communicating processes and mainly targets on telecommunication systems while ours uses a dataflow model, SDF [5], targeting for multimedia applications.

Figure 3 shows a dataflow graph of 2D DCT (discrete cosine transform) algorithm. It is a multirate dataflow graph with relatively high sample rates of 64 and 8. This algorithm can be mapped to various hardware implementations. A fully parallel implementation is taken in [9] by Ptolemy where eight resources of *DCT1D* block are created between two *transpose* blocks as shown in Fig. 4a. Note that eight *DCT1D* resources are activated at the same time after the *transpose* block produces 64 samples at once. And blocks are connected directly without buffer in-between. This parallel implementation results in the shortest processing delay but the largest hardware area, about 243,000 gates using 16 *DCT1D* resources in our experimentation.

In Meyr's approach [8, 16–18], the generated hardware structure has one-to-one correspondence to the dataflow graph where a separate hardware resource is allocated for each node. In Fig. 4b, one resource of *DCT1D* block is executed eight times sequentially to consume and produce 64 samples while one invocation of *DCT1D* block is assumed to consume eight samples at once[1]. If the *DCT1D* block is internally pipelined, eight invocations can also be pipelined. In this sequential implementation, a FIFO queue and a MUX are needed to accumulate 64 samples on the input and the output arcs of the *transpose* block respectively. But the hardware overhead is much smaller than that of parallel implementation. One difficulty of this approach is to generate numerous control signals whose timings are computed statically through rigorous graph analysis [17]. It has a serious restriction that all hardware blocks should have deterministic and fixed execution cycles for static timing analysis and controller synthesis.

Another sequential implementation approach is taken by Ade et al. [3], Dalcolmo et al. [19],

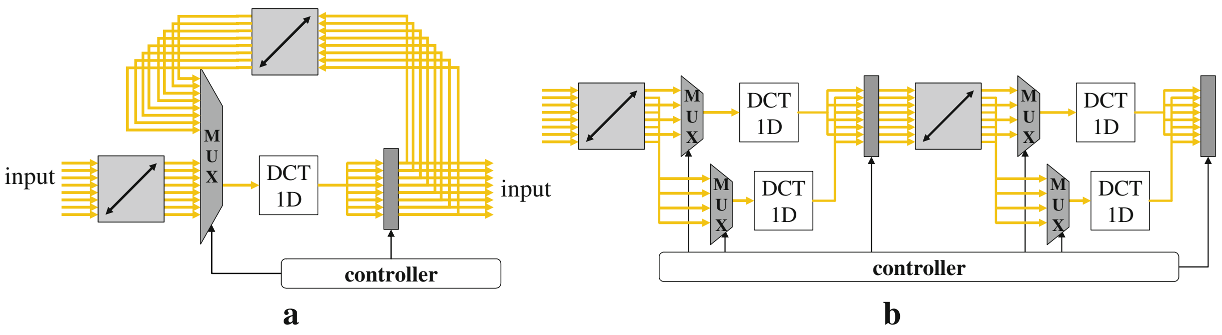


Figure 5. Hardware architectures with **a** one 1D DCT resource and **b** four resources.

Table 1. Comparison among the approaches of HW synthesis from DFG.

Approaches	Ptolemy	Meyr's	GRAPE	McAllister's	Proposed
Implementation of multi-rate specification	Parallel implementation	Sequential implementation	Sequential implementation	Parallel/sequential/hybrid implementation	Parallel/sequential/hybrid implementation
Resource allocation	Multiple-resource allocation	Single-resource allocation	Single-resource allocation	Multiple/single-resource allocation	Multiple/single/shared -resource allocation
Communication between blocks	Synchronous	Synchronous	Asynchronous (FIFO)	Asynchronous (FIFO)	Synchronous
Block control	Centralized control	Centralized control	Distributed control	Distributed control	Centralized control
Block execution time	Fixed	Fixed	Variable	Variable	Variable
Buffer optimization	N/A	Port retiming	Static minimum	N/A	Shift buffering/buffer sharing
Supported DFGs	SDF	SDF	SDF, CSDF	SDF, CSDF, MDSDF	SDF, FRDF

Lauwereins et al. [20] and in GRAPE system. Their approach is different from Meyr's in that each hardware block has its local controller and there is no need of complicated central controller design (Fig. 4c). A hardware block detects when it can be invoked by exchanging the control signals with its neighbor based on a certain hand-shaking protocol. In this example, *DCT1D* block is sequentially invoked eight times after the *transpose* block produces 64 samples, repeatedly exchanging hand-shaking control signals per invocation. Thus, this asynchronous communication incurs non-negligible runtime overhead while it is robust enough to allow non-deterministic block execution time. The hardware overhead comes from the FIFO buffers inserted between the blocks and the local controllers. Distributed control and asynchronous communication of Fig. 4c are not commonly used in an optimized ASIC design.

Figure 4a and b illustrate two extreme implementations of parallel and sequential operation. But, there are other implementation possibilities, some of which are displayed in Fig. 5. In Fig. 5a, a *DCT1D* resource is shared between two *DCT1D* nodes of Fig. 3. The *DCT1D* resource is invoked 16 times, 8 for column DCT and 8 for row DCT operation. It has the least amount of area with performance penalty. When better performance is required, the resources for column and row DCT are allocated separately and executed simultaneously in a pipelined fashion as already shown in Fig. 4b. In case the time constraint is tighter, more resources can be allocated as shown in Fig. 5b where two *DCT1D* resources are allocated to each *DCT1D* node. It is not fully-sequential nor fully-parallel, so it is called a hybrid implementation. Such resource sharing and hybrid implementation are also taken into account in the proposed approach. Thus, the proposed approach widens the hardware design space considerably compared with the previous approaches based on dataflow specification.

J. McAllister et al. [10] considered various implementations of a multi-dimensional SDF graph including hybrid implementation in targeting a reconfigurable device (e.g. FPGA). They assume asynchronous communication between nodes with a specific interface logic, called the *Control and Communication Wrapper (CCW).*

Table 1 summarizes the comparison between these approaches. The proposed approach is unique in the following aspects: First, the scheduling and hardware

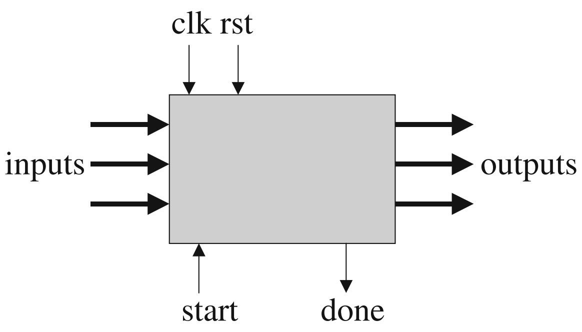


Figure 6. Input and output ports associated with a block with variable execution time.

synthesis is decoupled. Other approaches assume a fixed execution schedule of hardware components and synthesize the control logic to implement the schedule. On the other hand, a user may specify a schedule, and the hardware is automatically synthesized to implement the schedule in the proposed framework.

Second, it uses a centralized controller for efficient implementation, but allowing variable execution length of hardware components.

Buffer optimization on SDF arcs has been studied in Meyr's and GRAPE group. By retiming technique, Horstmannshoff and Meyr [8] minimized number of the shimming registers between nodes. The lower bound of arc buffers was investigated by Ade et al. [21] in under data-driven execution. Some approaches support the extensions of SDF: CSDF [22] for GRAPE and McAllister's, MDSDF [23] for Mcallister's, and FRDF [6] for our approach.

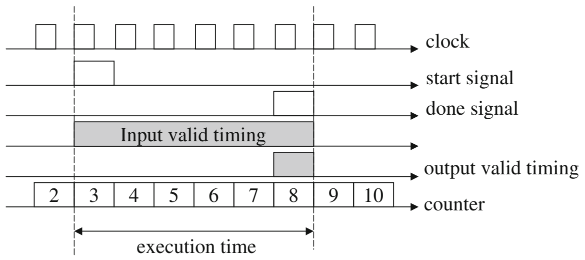
3. Block Types and Control Signals

In the proposed methodology, the complexity of a block definition is not restricted as long as the block follows the SDF semantics in which the block is triggered only when all input ports have enough data samples. It is important to note that the concept of a "data sample" in the dataflow model is different from that of an "event" that usually means the change of signal level on a wire. The arrival of a new sample should be notified by the predecessor block or by the controller based on the block schedule information. By preserving the SDF semantics in the generated hardware, we can guarantee the equivalence between the synthesized system and the dataflow specification in terms of functionality: So the synthesized hardware is claimed to be *correct by construction*.

To preserve the SDF semantics, we insert a buffer as a glue logic on every arc between two blocks. The buffer is latched with new output samples from the source block only after the block completes its execution. The destination block may read the valid data samples during the whole execution period. Thus the block needs no internal buffer to latch the input signal values. Buffer insertion strategy is also adopted in Sharp and Mycroft's [24] work on SAFL language which specifies hardware behavior in a higher level abstraction. While they do not assume dataflow model, they observed that higher level structuring mechanism needs storage elements not wires in HW synthesis.

The arc buffers are automatically generated and managed. The buffer management scheme is closely related with the types of blocks. We classify functional blocks into three different types based on the timing requirement of the internal behavior. The first type is *combinational logic* that does not include any internal register. Since we allow multi-cycle combinational logic, we compute the execution latency in terms of global clock cycles and trigger the output buffer load enable signal after the execution latency once the block is triggered by new input samples.

The second type is a *sequential logic with a fixed execution time*. Since a sequential logic includes internal registers, three additional control signals (*start*, *reset*, and *clock* signals) are provided for the timing management of the internal registers. The *reset* signal is triggered in the initialization phase of the synthesized hardware. After the *start* signal is



- Start time = 3, End time = 8
- Execution time = End time – Start time + 1 = 6(cycles)

Figure 7. An example timing diagram of the control signals.

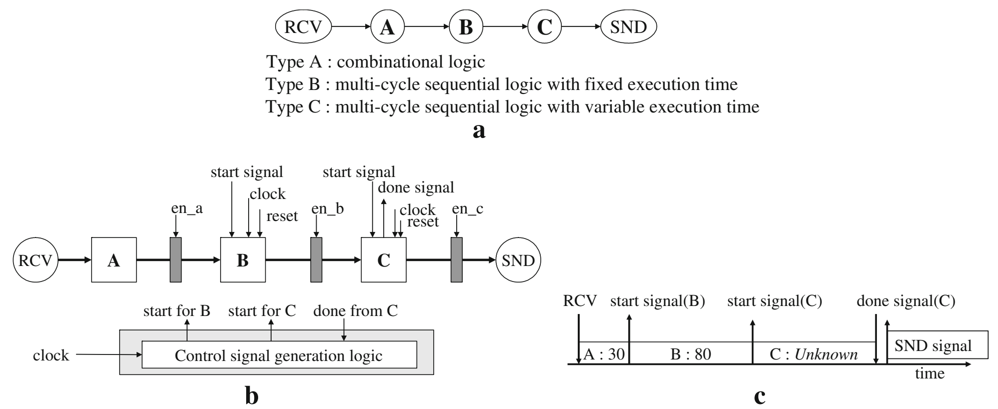


Figure 8. **a** An example DFG with various types of blocks, **b** synthesized hardware structure, and **c** the expected signal timings.

enabled, a fixed number of *clock* signals are counted to trigger the output buffer load enable signal. In case the minimum cycle time of a block is larger than the global clock period, the *clock* signal is obtained by down-sampling the global clock.

The third type is a *sequential logic with a variable execution time*. Then we need another control signal, *done*, to indicate the completion of block execution: When the *done* signal is enabled, the output buffer load enable signal is triggered. Figure 6 shows the resultant input and output ports associated with this type of block and Fig. 7 illustrates an example of a timing diagram for the control signals. Since it is the most general type, it can be used for general legacy hardware IP. A third-party IP whose protocol is known to designer can be abstracted by the wrapper that translates IP specific protocol to this type. The interrupt signal or a register value change that denotes the end of IP execution, for example, can be translated into the assertion of *done* signal. In this way, the proposed framework allows the use of legacy hardware IP blocks as long as there are corresponding dataflow nodes in the initial specification.

```
# resource allocation table
Transpose      2
DCT1D          2
# resource mapping & schedule information
# (instance name, resource number, start, duration)
# loop ( loop count, start, loop period)
Transpose_0    0      0      1
Loop    8    1    2  {
          DCT1D_0    0      0      2
}
Transpose_1      1     17      1
Loop    8     18     2 {
          DCT1D_1     1      0      2
}
```

Figure 9. Schedule information for the architecture of Fig. 4(b).

Figure 8a shows a simple example that consists of all three types of blocks. The block types should be explicitly specified by the designer. Then the glue logics between blocks and the central controller are automatically synthesized, and integrated with the library blocks to result in the final architecture as shown in Fig. 8b. The design of controller is explained in the next section. In Fig. 8c the expected timing of control signals is drawn. Here, we assume that execution times of blocks A and B are 30 and 80 time units respectively, while the execution time of node C is unknown; the timing of *done* signal for block C is determined at run-time.

The proposed technique discussed in this section can be summarized as follows:

1. Two adjacent blocks are communicated with each other through arc buffers. So the communication between blocks is managed simply by defining the timing of the buffer control signals. However, it implies that we add a buffer between two combinational logic blocks. Such extra buffer can be reduced by post-optimization phase in the proposed methodology, which will be discussed in Section 6.
2. The start signal is triggered to a sequential logic only after all input buffers complete latching of new data samples to satisfy the SDF semantics. If

a block has multiple input arcs, it may lengthen the critical path length of a block. Relaxing the strictness of the SDF semantics for performance optimization is a future research subject.

3. The data samples on the input buffers remain valid at all times so that no internal buffering to latch the input signals is needed inside the block.

4. Proposed Hardware Synthesis Technique

4.1. Schedule Information Structure for H/W synthesis

The schedule information is obtained from the partitioning step in the proposed approach. Figure 9 shows an example schedule information for the architecture of Fig. 4b. The schedule information consists of two parts. One is resource allocation table and the other is mapping/schedule information.

Resource allocation table is simply a set of pairs, {resource type name, number of resources}. There are two resources of *transpose* blocks and two of *DCT1D* blocks in Fig. 9. Mapping/schedule information defines the timing information of each *instance* of nodes. It also defines the allocated resource to the instance. It may be grouped by a loop to make a hierarchical representation. The syntax of the schedule information can be concisely represented by the following Backus Naur form (BNF) representation.

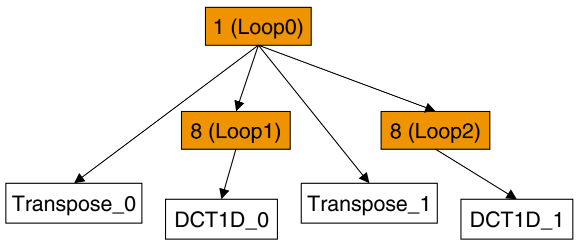


Figure 10. Schedule information as a tree form: Leaf nodes and internal nodes stand for the nodes and loop iteration numbers respectively.

For example, "*transpose*_1 1 17 1" indicates that *transpose_1* node uses the second resource (resource number=1) between two *transpose* resources and its start timing is 17.

If a designer wants to generate a hardware structure like Fig. 5a, where two *DCT1D* blocks shares one hardware resource of *DCT1D*, he or she only has to modify the number of *DCT1D* resource to 1 and the mapped resource number of *DCT1D_1* instance to 0 as follows.

DCT1D 1

DCT1D_1 **0** 0 2

It means that only one *DCT1D* resource is allocated and all eight instances of *DCT1D_1* node are also mapped to this resource. The time unit used to specify the start time and execution length is a clock cycle.

```
<schedule information> ::=
<allocation table>
<mapping_schedule information>
```

```
<allocation table> ::= set of <allocation item>
<allocation item> ::=
<resource type name> <number of resources>
```

```
<mapping/schedule information> ::=
set of <mapping_schedule item>

<mapping_schedule item> ::=
<instance mapping_schedule> | < loop mapping_schedule>

<instance mapping_schedule> ::=
<node name> <mapped resource number> <start timing> <execution time>

<loop mapping_schedule> ::=
loop    <loop count> <start timing> <loop period>
{ <mapping_schedule information> }
```

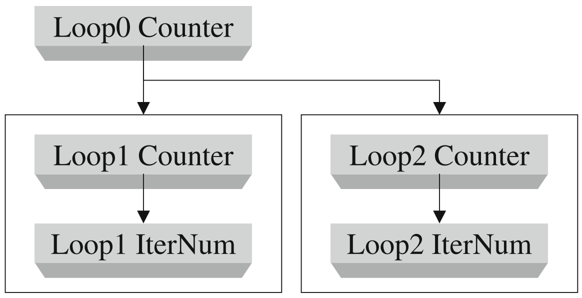


Figure 11. Hierarchical loop control structure.

The schedule information in Fig. 9 contains two schedule loops. Since each loop has a local counter, the start time of the first block in a loop is set to 0. Loop 8 1 2 {DCT1D_0 0 0 2 } means that all eight instances of *DCT1D_0* node use the first hardware resource of *DCT1D* and the resource is executed eight times consecutively with execution time equal to two cycles. Note that the repetition period is two while the loop starts at one. The schedule information associated with a *DCT1D* block of Fig. 5b can be represented as Loop 4 1 2 {DCT1D_0 0 0 2, DCT1D_0 1 0 2}. It means that first, third, fifth, and seventh instances are mapped to the first resource and the others are to the second resource. The entire schedule is also regarded as a loop since DFG is repeatedly executed as an infinite loop.

4.2. Counter-based Controller

The schedule information of Fig. 9 can be organized as a tree data structure as shown in Fig. 10. Since the entire schedule is regarded as a loop, Loop0 is used for top-level control with loop count equal to one and the others are numbered in sequence. We synthesize the loop control hardware structure in the same fashion as the schedule data structure. As shown in Fig. 11, a loop counter is allocated for each loop and an iteration number counter is created for nested loops in order to count the loop iterations. Each loop counter and iteration number counter are controlled by their parent counter. The timing diagram in Fig. 12 shows the relationship among the counter values. Loop1 counter starts when Loop0 counter value becomes 1. The counter value increases by 1 at every clock cycle and Loop1 iteration number counter increases at the end of an iteration of Loop1.

In case a loop contains a block that has varying execution time, the loop counter is stalled at the end of scheduled time of the block until it receives the *done* signal from the block. Then, the parent loop should check the done signals of the child loops with variable execution time recursively. Figure 13 shows an example in which Loop0 checks the done signals of Loop1 and Loop2 at their expected end times. If a done signal is not enabled, the counter waits until it is enabled.

4.3. Buffer Allocation

In the proposed technique, buffers between dataflow nodes are automatically allocated and connected to hardware resources. It should be noted that each port of dataflow hardware block may consume and produce multiple data samples at once. Consider a simple example in Fig. 14a. If we use only one hardware resource for each node, the schedule may be AABAB or (3A)(2B) where (3A) means A block is executed three times consecutively. In case the schedule is AABAB, the minimum required buffer size on the arc becomes four. If we use a looped schedule of (3A)(2B), the minimum required buffer size is increased to six. The relationship between scheduling and buffer size has been addressed in many researches [25] for software code generation. Buffer size requirement is also a key factor for hardware code generation. If we use maximum hardware resources for parallel execution using three resources for A and two resources for B in the example of Fig. 14a, the required buffer size should be six. Figure 14c illustrates buffer allocation and signal connection between buffers and blocks in this case. The required buffer size in case of maximum parallel execution becomes total number of sample exchange (TNSE) of the arc [25]. Given an dataflow arc a, we denote the source node and sink node of a by src(a) and snk(a) respectively. Also, $p(a)$ and $c(a)$ denote the number of data samples produced onto a by src(a) and consumed from a by snk(a) respectively. Then,

$$\begin{aligned} TNSE(a) &= \mathbf{q}(\mathrm{src(a)}) \times \mathrm{p}(a) \\ &= \mathbf{q}(\mathrm{snk(a)}) \times \mathrm{c}(a) \end{aligned} \tag{1}$$

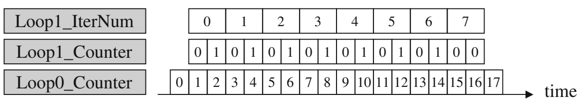


Figure 12. Timing diagram of loop control counters.

```
constant Last_CounterValue : integer := 34;
…
Loop0_counter : process(clock, reset)
begin
      if reset = '1' then
            CounterValue <= -1;                              -- inactive state
      elsif rising_edge(clock) then
            if CounterValue = Last_CounterValue then
                  CounterValue <= 0;                         -- to the next iteration
            elsif CounterValue = 16 and Loop1_done_reg = '0' then
                  CounterValue <= CounterValue;              -- waiting
      elsif CounterValue = 33 and Loop2_done_reg = '0' then
                  CounterValue <= CounterValue;              -- waiting
      else
                  CounterValue <= CounterValue + 1;          -- advancing
            end if;
      end if;
end process;
```

Figure 13. VHDL code for counter controller.

in which **q**(n) is the repetition count of a node n in an iteration of a schedule.

Basically, we allocate as many buffers as TNSE to guarantee safe buffering in all cases. However, high sample rates may cause huge buffer overhead. If a dataflow graph is scheduled in a loop fashion, we can reduce the buffer size utilizing the *loop factor*. Consider an example with high sample rates in Fig. 15. The repetition count of each node and TNSE of each arc can be calculated easily from the specified data sample rates. Fractional rate (the producing rate of node D is 1/5) will be discussed in the next section. And we assume that the schedule is A10(B5(CD))E. We define the *loop factor* of an arc a, LF_a, as the multiplication of loop iteration numbers of common ancestors of two end nodes. Figure 15b depicts the tree data structure of Fig. 15a where the leaf nodes denote SDF nodes and the internal nodes are labeled by loop iteration numbers as explained in Fig. 10. LF_{BC} is calculated by multiplication of 1 and 10, since the common ancestors of nodes B and C are root node(1) and the internal node which is labeled by "10". Then the required buffer size of arc a, BS_a, is computed with the following equation:

$$BS_a = TNSE(a)/LF_a + D(a) \qquad (2)$$

where $D(a)$ denotes the number of delay elements (or initial samples) on arc a.

In case delay elements exist on dataflow arcs as in Fig. 16a, as many buffers as the number of the delay elements are prepended at the head of the buffer array. The last buffer is connected to the delay

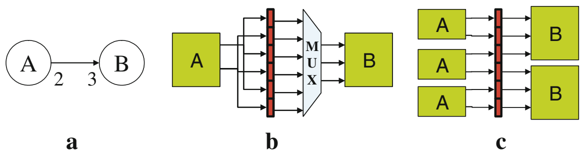


Figure 14. Buffer allocation and signal connection for a simple example: **a** a simple SDF specification, **b** implementation of single resource allocation case, and **c** implementation of multiple resource allocation case.

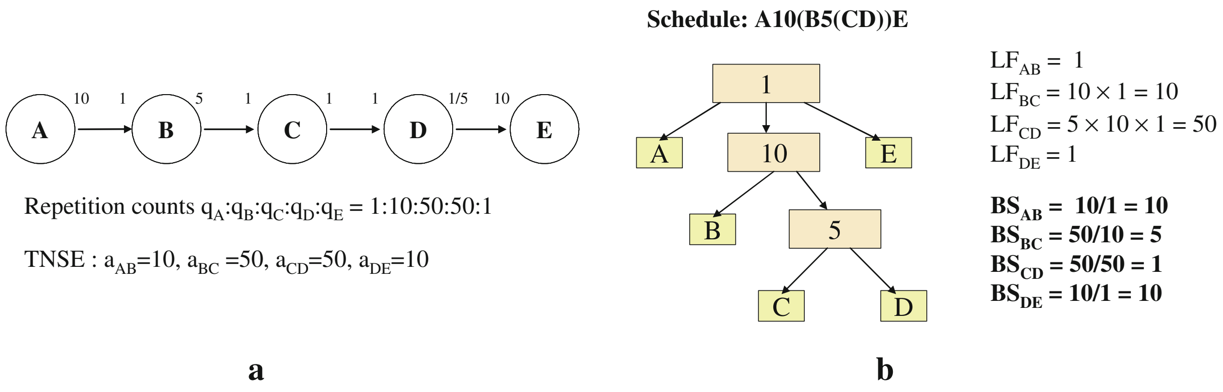

Schedule: A10(B5(CD))E
10 1 5 1 1 1 1/5 10
A B C D E
Repetition counts $q_A:q_B:q_C:q_D:q_E$ = 1:10:50:50:1
TNSE : a_{AB}=10, a_{BC} =50, a_{CD}=50, a_{DE}=10
1 10 5
LF_{AB} = 1
$LF_{BC} = 10 \times 1 = 10$
$LF_{CD} = 5 \times 10 \times 1 = 50$
LF_{DE} = 1
BS_{AB} = 10/1 = 10
BS_{BC} = 50/10 = 5
BS_{CD} = 50/50 = 1
BS_{DE} = 10/1 = 10
a
b

Figure 15. **a** A Dataflow graph with high sample rates and **b** its schedule data structure.

buffers to make a ring buffer as shown in Fig. 16b. While the output signals from the source node are connected to the empty arc buffers, the input signals to the sink node are connected starting from the delay buffers. The remaining data samples after each loop iteration are moved to the delay buffer area before starting the next iteration.

In case the size of data sample is large, using a memory is preferable to arc buffers in terms of hardware area and cost. So we map the arc buffers to the memory for large size data samples. In this case, the address or pointer is transferred for synchronization and the real data samples are delivered through memory using memory access interface as shown in Fig. 17. To use memory interface, the block writer should set the memory attribute of ports. This attribute enables automatic generation of memory interface logic that connects to the memory or bus. The block programmer should make memory access hardware code directly using the provided interface of which details are explained in [4].

4.4. *Control H/W Structure and VHDL Codes*

The proposed hardware code generation technique is to generate the corresponding hardware architecture from the given schedule/mapping information. For example, it automatically generates the HW architecture of Fig. 4b from the schedule/mapping information of Fig. 9. We generate an RTL code that includes instantiated hardware resources from block library, counter-based controller structure as presented in Section 4.2, and glue logic such as MUXes and registers considering resource sharing.

To support resource sharing, we construct data structures for MUX control and *start* signal management for the shared resources. In order to control the input MUX of a resource, we store the information on which nodes and which instances are mapped to the resource and when each invoca-

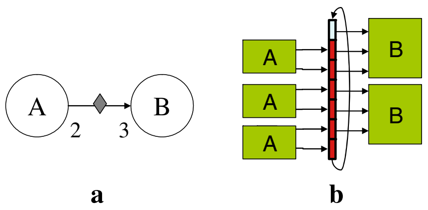


Figure 16. Delay element and buffer connection: **a** specification and **b** implementation.

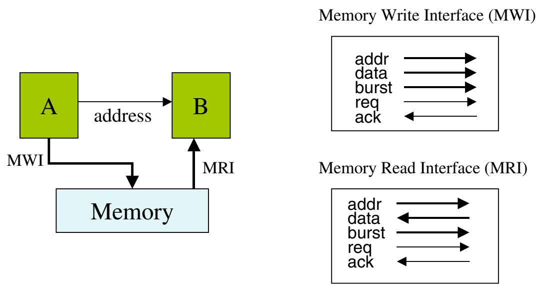


Figure 17. Memory interface generation for large data sample. Control H/W.

```
-- <input mux code>
<resource input signal name> <=
              <input buffer_1 name> when Counter >= <start time_1> and Counter <= <end time_1> else
              <input buffer_2 name> when Counter >= <start time_2> and Counter <= <end time_2> else
              ...
              <input buffer_N name>;
-- <start signal code>
<start signal name> <=
              '1' when Counter = <instance_1 start time> else
              '1' when Counter = <instance_2 start time> else
              ...
              '1' when Counter = <instance_N start time> else
              '0';
```

Figure 18. VHDL code templates for input MUX and start signal management.

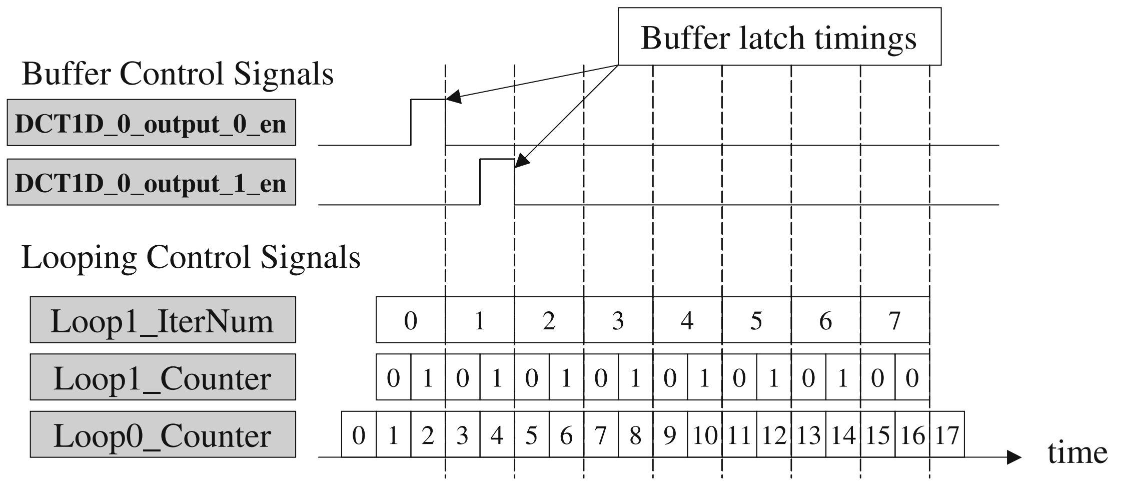
tion starts and ends. Data structures are organized per each resource by analyzing the given schedule information. Figure 18 shows the VHDL code templates generated from these data structures. MUX is controlled by a range of counter value since a MUX should hold the valid input signal during the execution of its target resource. In case the schedule has a loop, the conditional statement becomes more complicated considering the loop iteration number as illustrated in the VHDL code of Fig. 20. The VHDL code in Fig. 18, however, has a simple conditional statement using only one counter value. As shown in Fig. 18, code for the *start* signal management is simpler since only start timings of instances are needed.

We also need to manage the output buffer *load enable* signals to store output data samples from the shared resource. The information on when to latch the signal and from which resource is directly obtained from the schedule file. We can determine when to latch the output signal just by adding the start timing and its execution time. Figure 19 shows a simple VHDL code template for output buffer management.

Figures 20 and 21 show the timings of control signals and the resultant hardware structure described in this section for the *DCT1D* block of Fig. 4b. The control signals in these figures such as *DCT1D_0_output_0_en* and *DCT1D_res0_sel* are not defined explicitly in the generated RTL code. They are generated implicitly

```
-- <output buffer code>
process(clk, rst)
begin
          if rst = '1' then
                    <output buffer signal name> <= (others => '0');
          elsif rising_edge(clk) then
                    if Counter = <latch timing_1> then
                              <output buffer signal name> <= <resource output signal_1 name>;
                    elsif Counter = <latch timing_2> then
                              <output buffer signal name> <= <resource output signal_2 name>;
                    ...
                    elsif Counter = <latch timing_N> then
                              <output buffer signal name> <= <resource output signal_N name>;
                    end if;
          end if;
end process;
```

Figure 19. VHDL code template for output buffer management.



Input Mux Control VHDL Code

```
DCT1D_res0_input <=
   Transpose_0_output_0 when Loop1_IterNum = 0 and Loop1_Counter >= 0 and Loop1_Counter <= 1 else
   Transpose_0_output_1 when Loop1_IterNum = 1 and Loop1_Counter >= 0 and Loop1_Counter <= 1 else
   …
   Transpose_0_output_6 when Loop1_IterNum = 6 and Loop1_Counter >= 0 and Loop1_Counter <= 1 else
   Transpose_0_output_7;
```

Figure 20. Timing diagram of control signals and VHDL code for input MUX of DCT1D in Fig. 4(b).

from the RTL code. As illustrated in Fig. 21, all control signals for MUXes and buffers are determined by the counter values of the loop controller.

5. Fractional Rate Dataflow Specification for More Efficient Implementation

Memory efficient code synthesis from synchronous dataflow models has been an active research subject to reduce the gap in terms of memory requirements between the automatically synthesized code and the manually optimized code of software [25, 26]. These works, however, minimize the buffer requirements by optimal scheduling, not by overcoming the limitations of dataflow semantics. Fractional rate

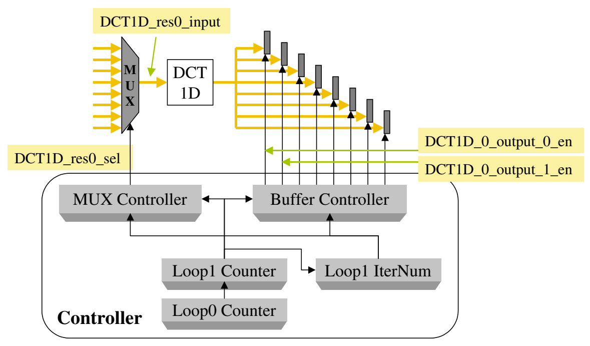


Figure 21. The hardware structure of buffers, loop controllers, and MUXes for the DCT1D block in Fig. 4b.

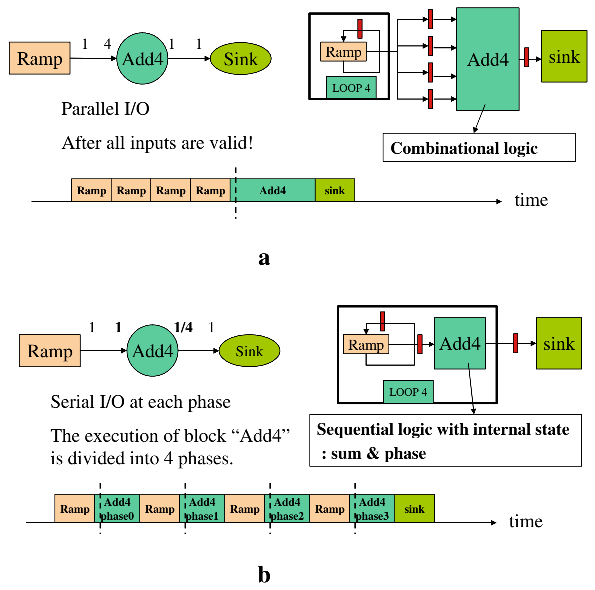

Ramp
1
4
Add4
1
1
Sink
Parallel I/O
After all inputs are valid!
Ramp
LOOP 4
Add4
sink
Combinational logic
Ramp Ramp Ramp Ramp Add4 sink
time
a
Ramp
1
1
Add4
1/4
1
Sink
Serial I/O at each phase
The execution of block "Add4" is divided into 4 phases.
Ramp
Add4
sink
LOOP 4
Sequential logic with internal state : sum & phase
Ramp Add4 phase0 Ramp Add4 phase1 Ramp Add4 phase2 Ramp Add4 phase3 sink
time
b

Figure 22. **a** SDF vs **b** FRDF implementation of *Add4*.

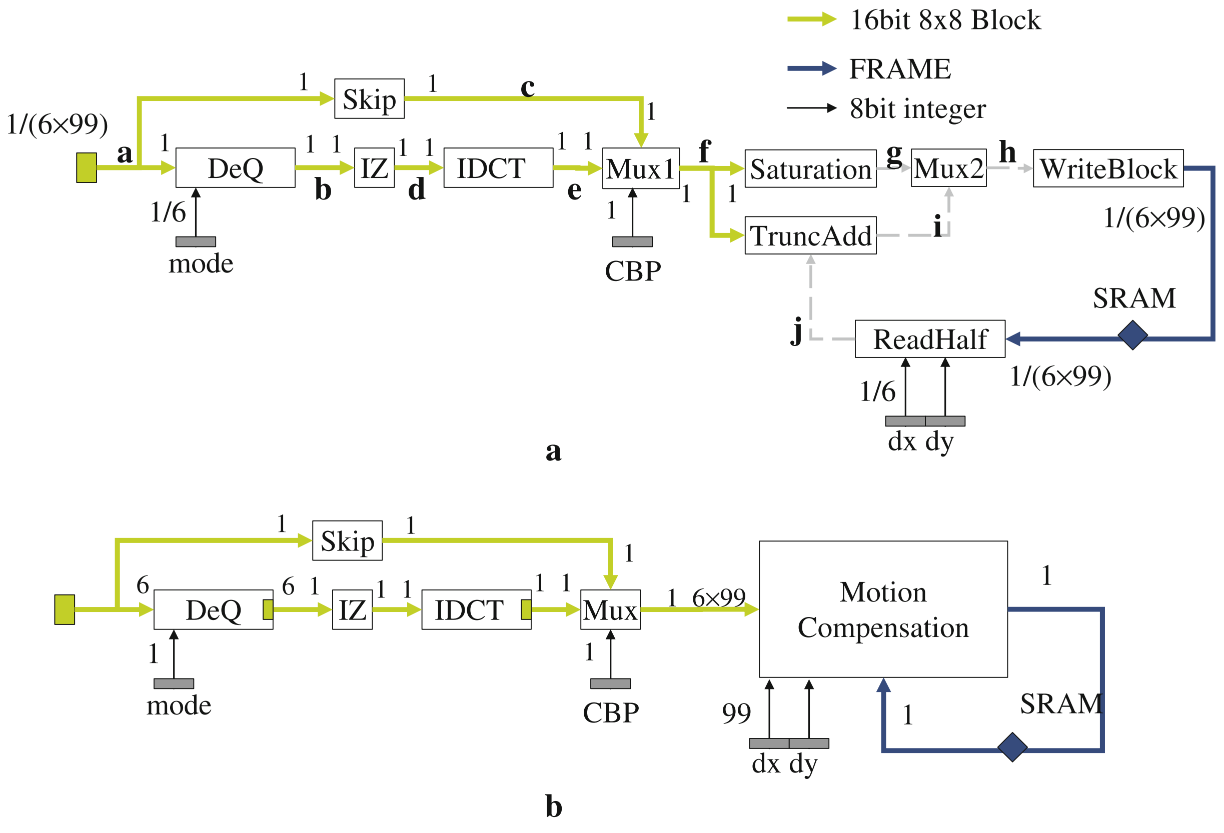


Figure 23. **a** An FRDF and **b** an SDF specification of a subset of H.263 decoder algorithm.

data flow (FRDF) model is proposed in [6] to overcome the limitations by allowing fractional sample rates in the dataflow specification.

Composite data types, such as video frame or network packet, are used extensively in multimedia applications, and become the major consumer of scarce memory resource. Existent dataflow models have inherent difficulty of efficiently expressing the mixture of a composite data type and its constituents: for example, a video frame and macroblocks. A video frame is regarded as a unit of data sample in integer rate dataflow graphs, and should be broken down into multiple macroblocks explicitly consuming extra memory space.

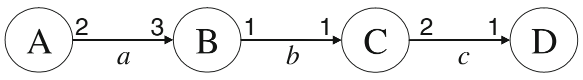


Figure 24. A simple example.

In the FRDF model, a macroblock is regarded as a fraction of a video frame.

The concept of fractional rate can be used not only for composite data type but also for atomic data type such as integer or float. An example is shown in Fig. 22b, in which the producing rate of *Add4* block is 1/4. This means that *Add4* block produces one data sample per every four invocations, not 1/4 data sample per invocation. This concept is useful in hardware code generation since the synthesized hardware area is sensitive to size of buffer register.

In Fig. 22a the SDF model forces the proposed technique to allocate four input buffers for the *Add4* block to store output samples from the looped execution of the *Ramp* block. In this case, the *Add4* block may be implemented to a combinational logic of 4-input adder. On the other hand, Fig. 22b represents the equivalent FRDF graph and its hardware implementation. Fractional sample rate 1/4 of the output port of the *Add4* block implies that only

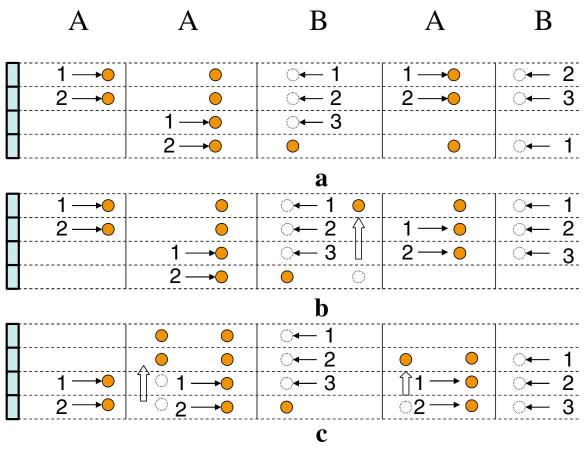


Figure 25. Indexing patterns of arc a of Fig. 24: **a** traditional indexing, **b** *read-fixed indexing*, **c** *write-fixed indexing*.

one output sample is produced after four invocations of the *Add4* block. In other words, each invocation consumes one input sample, but every fourth invocation produces one output sample. This specification generates a sequential implementation with an adder. In Meyr's approach, the *Add4* block will be invoked four times and only one buffer is allocated on the input arc even if the input sample rate of Add4 is 4. In fact Meyr's approach assumes non-strict execution with serial I/O on a multi-rate port. The FRDF specification achieves the same implementation as Meyr's.

Another example of FRDF specification is shown in Fig. 23. This figure specifies a subset of H.263 decoder algorithm. It includes *Dequantize*, *IDCT*, and motion compensation logic. In this specification, array types and user-defined types are used for data samples and VHDL code generation for their signal definitions and port mappings are automatically performed. Since memory access logic for a frame memory is needed during motion compensation, the attributes of ports that access frame memory are set to *SRAM*. This attribute generates ports for the memory access and lets the frame data sample be transferred by only its address or pointer. Note that fractional rates are specified for frame data and some parameter data samples such as *mode*, *dx*, and *dy* to save buffer area.

Originally, the input data sample rate of motion compensation block is 6×99 for QCIF format as shown in Fig. 23b since motion compensation block needs 99 macroblock inputs for one frame output. Such specification, however, results in large buffer overhead of 6×99 input buffers for MC block. FRDF specification as depicted in Fig. 23a can save buffer area significantly in case of multimedia applications with high sample rates. The experimental results of this example are presented in Section 7.

6. Buffer Optimization

As shown in the previous section, FRDF model allows us to reduce the buffer size significantly compared with the SDF model. The FRDF model uses the notion of a fractional sample: for example, an image frame is decomposed into 99 fractional samples, macroblocks(16×16) in QCIF(176×144) format. By specifying in the FRDF model, we could reduce the number of frame-size buffers in the synthesized hardware. Nonetheless, Jung and Ha [27] reports that the automatically synthesized hardware has about 50% overhead in buffer area compared with manually optimized design. In this section, we present how to reduce the buffer

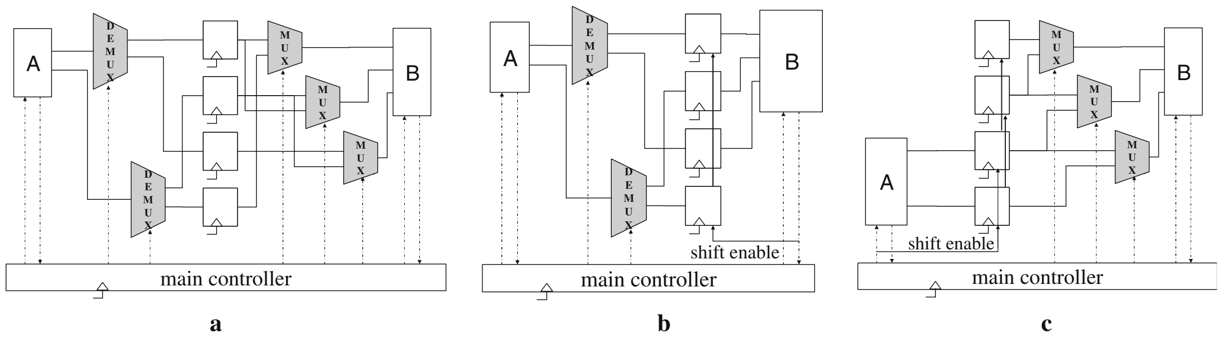


Figure 26. Hardware implementations of Fig. 25.

	A	A	B	A	B	C	C	D	D	D	D
a	2	4	1	3	0	0	0	0	0	0	0
b	0	0	1	1	2	1	0	0	0	0	0
c	0	0	0	0	0	2	4	3	2	1	0

Figure 27. Number of remaining samples in arcs following schedule AABABCCDDDD.

overhead further to make it close to the manually optimized design.

Buffer size optimization has been an important research issue in automatic software synthesis from dataflow specification. Given a dataflow graph and a predetermined node schedule, both global buffer minimization [28] and buffer merging technique [29] analyze the lifetimes of arc buffers to share the buffers that do not overlap with each other. Recently shift buffering technique is devised to minimize the buffer size in case no linear buffering is possible or modulo addressing is needed to address the buffer entries [30].

Let's take a simple example of buffer assignment in Fig. 24. If we assign buffers to each arc as many as *TNSE* by Eq. (1) in Section 4.3, the number of buffers assigned to the whole graph is 6+2+4=12. Note that this buffer allocation policy supports any schedule of hardware execution because the buffer is big enough to keep all the samples generated during an iteration. If a dataflow graph is scheduled in a loop fashion, we may reduce the buffer size exploiting the loop factor as explained in Section 4.3. Under the schedule 3A2(BC2D), the number of buffers is reduced to 6/1+2/2+4/2=9 by Eq. (2) in Section 4.3.

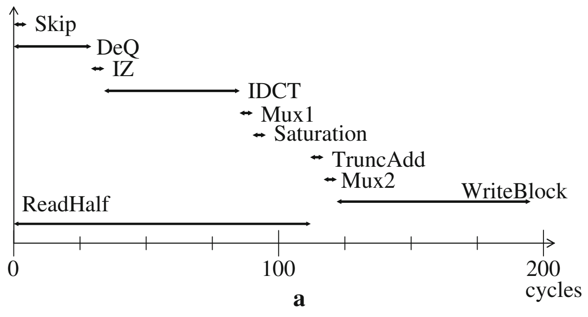
6.1. Shift Buffering

For arc *a* in Fig. 24 we can reduce the buffer size if we consider the execution schedule. In case the execution schedule of nodes A and B is AABAB, the maximum number of sample accumulated on the arc is only four which is less than six that is *TNSE* of the arc. Suppose we allocate four buffers on arc *a*. Then the buffer indexing pattern becomes complicated as Fig. 25a illustrates: the second invocation of node B requires wrapped-around buffer indexing scheme. To enable this way of indexing, three multiplexers are attached between node B and the buffer, and two demultiplexers between node A and the buffer as shown in Fig. 26a.

To avoid such complicated indexing, we devise shift-register based buffering, shortly *shift buffering*. It simplifies the control logic of buffer management. Shift buffering is divided into two categories: *read-fixed indexing* and *write-fixed indexing*.

Read-fixed indexing fixes the read pointer of the consumer. In our example, consumer node B always read the data samples from the same buffer position as shown in Fig. 25b. To manage this scheme correctly, buffers need to be shifted by three in this case, after every firing of consumer. It means that we do not need to attach MUXes to consumer's input ports any more.

Figure 26b illustrates the implemented hardware in the scheme of *read-fixed indexing*. The read ports of consumer node B are hard-wired to some positions of buffer. The last buffer is shifted by three steps as the shift enable signal is granted. In the scheme, buffer shift should happen after every firing of consumer node, and that is why the *done* signal of consumer node B is fed to the *shift enable* port.



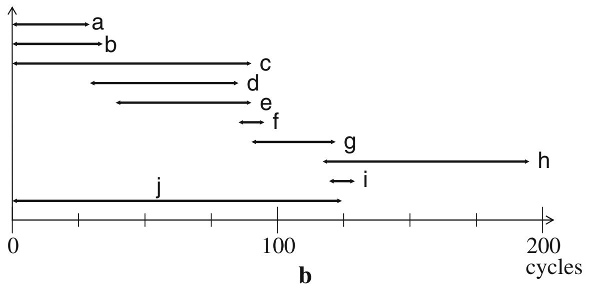


Figure 28. Life-time analysis of H.263 decoder: **a** life time of each node, **b** life time of each arc.

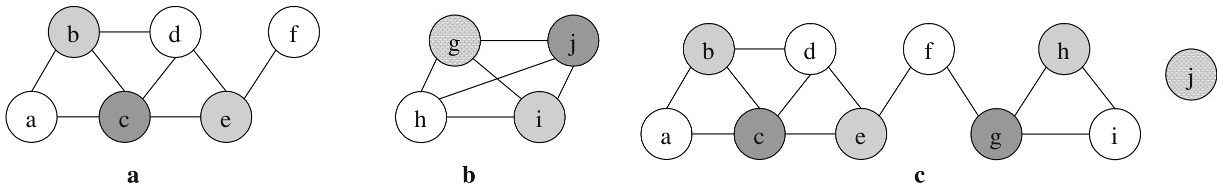


Figure 29. Buffer allocation by minimum graph coloring: **a** data type, 64 array of 16 bits; **b** data type, 64 arrays of 8 bits; **c** merged data types.

Shift buffering enables us to minimize the buffer size without severe hardware area increase. The minimum buffer size is nothing but the maximum number of data samples accumulated in the buffer during an iteration. We name that number as *maximum number of remaining sample*, MNRS. Then the total size of buffer becomes

$$BS(G) = \sum_i (MNRS(\alpha_i)) \tag{3}$$

To reduce the hardware area further, we can apply above techniques in a mixed way. For instance, as you see in Fig. 26, we can eliminate three MUXes by *read-fixed indexing* or two DEMUXes by write-fixed indexing. Since the hardware complexity of MUX is similar to that of DEMUX, we can get the better result from read-fixed indexing. So, we apply an appropriate indexing policy for each arc. Here is the pseudo-code of a simple algorithm which determines the buffer size and indexing policy of each arc. The buffer management is relatively easy when $p(i)$ or $c(i)$ can be divided by the other. Otherwise, we can use *shift buffering* to get benefits in terms of buffer size and glue logic overhead.

```
1: for all arc $i$ :
2:     if  $p(i) \% c(i) = 0$ or $c(i) \% p(i) = 0$ then
3:         allocate buffers as many as $TNSE$
4:         apply traditional buffer indexing
5:     else
6:         if  $p(i) > c(i)$ then
7:             allocate buffers as many as $MNRS$
8:             apply write-fixed indexing
9:         else
10:            allocate buffers as many as $MNRS$
11:            apply read-fixed indexing
12:        end if
13:    end if
14: end for
```

6.2. Buffer Sharing

As you see in Fig. 27, the number of live data samples does not exceed four always in the schedule, AABABCCDDDD. Traditional buffer allocation policy, however, synthesizes at least two times more than optimal buffer size as in the way explained

Table 2. Experimental results: two-dimensional DCT.

		Latency			
Design type (number of DCT1D resource)	Area (gates)	Clock period (ns)	Cycles	Total (ns)	Throughput (sample/ms)
Manual (1), Fig. 5a	31,431	20	32	640	1,562.5
Auto (1), Fig. 5a	38,265	20	34	680	1,470.6
Auto (2), Fig. 4b	53,252	20	34	680	2,941.2
Auto (4), Fig. 5b	83,130	20	18	360	5,555.6
Ptolemy (16), Fig. 4a	242,944	20	4	80	12,500.0
GRAPE (2), Fig. 4c	79,832	20	52	1,040	2,083.3

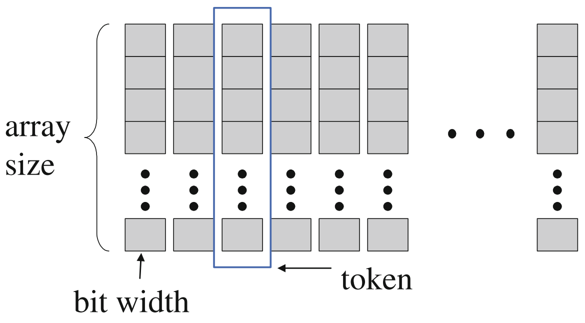


Figure 30. Type definition of an arc buffer.

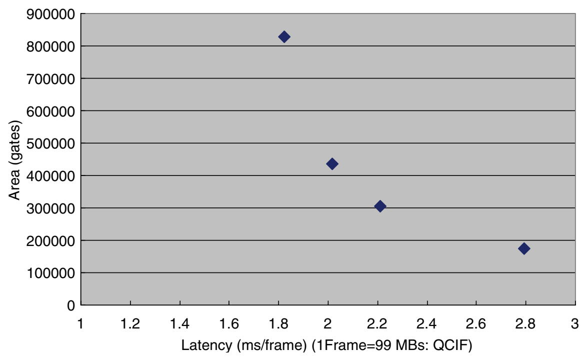


Figure 32. Area-delay graph of Table 3.

above. This is the major factor that makes hardware much larger.

Buffer sharing technique is to share arc buffers only when their life times are not overlapped during an iteration of the schedule. Figure 27 shows the number of remaining samples in each arc of Fig. 24 following schedule AABABCCDDDD. In Figure, we can identify that buffers on arc *a* and arc *c* do not overlap in their life times so can be shared. If they are shared, the required buffer size is reduced to

$$\begin{aligned} BS(G) &= MNRS(b) \\ &\quad + Max(MNRS(a), MNRS(c)) \\ &= 2 + 4 = 6 \end{aligned} \tag{4}$$

In general, buffer sharing technique reduces buffer size as the following equation: for all shared buffer indices *j*, and all arc indices *i* in each shared buffer,

$$BS(G) = \sum_j \left(\underset{i}{Max}(MNRS(\alpha_{ji})) \right) \tag{5}$$

The life times of arcs are analyzed using schedule information. A life time may not be continuous sometimes which makes proposed technique different from conventional register allocation algorithm in compiler. The life time of an arc is not the whole duration from the first writing to the final reading, but only the active(which means at least 1 or more tokens exist in it) part of life time based on static schedule.

We explain the proposed buffer sharing technique with a complicated example of Fig. 23a in which each arc except the right-most one is named by an alphabet. The right-most arc is implemented in SRAM because it is too bulky to be mapped to a register array. Note that the arcs that have multiple readers (arcs *a* and *f*) are shared basically. Solid lines and dashed lines represent 64 arrays of 16 bits and 64 arrays of 8 bits, respectively. Figure 28a illustrates the schedule information from which we obtain the life time of each arc as shown in Fig. 28b.

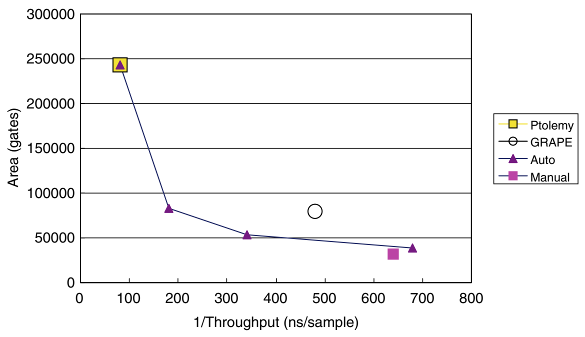


Figure 31. Area-delay graph of Table 2.

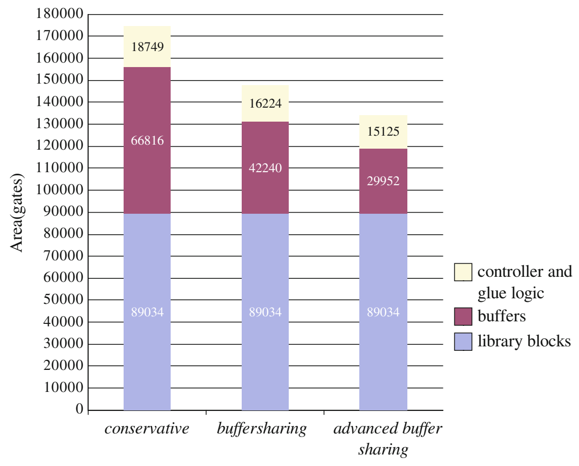


Figure 33. Experiments of buffer sharing.

Table 3. Experimental results: H.263 decoder.

	HW area (gates)				
HW implementation	Library block	Buffers	Glue logics	Total	Performance: QCIF (ms/frame)
1 resource	89,034	66,816	18,749	174,599	2.79
2 resources	149,878	116,480	39,969	306,327	2.21
3 resources	210,722	166,144	60,275	437,141	2.02
6 resources	393,254	315,136	120,251	828,641	1.82

The problem of finding the optimal buffer sharing is converted to minimum graph coloring. First we consider sharing buffers of the same type only. Figure 29a shows how the graph is colored by three colors where six buffers (of 64 arrays of 16 bits) are reduced three. The buffers of 64 arrays of 8 bits, however, cannot be reduced further because it is a complete graph as shown in Fig. 29b.

For further buffer sharing, we compare the buffer sizes among different data types. We define the arc type as a tuple <array size, bit width, sample count> as shown in Fig. 30. For example, the type of arc *c* in Fig. 23a is defined as <64, 16, 1>. We can share two arcs when all of three terms of a buffer is bigger (or equal) than (or to) the other. Since arc *c* and arc *h* are the case, they are sharable. Figure 29c illustrates the final buffer sharing. Here all the edge associated to *j* is omitted because *j* is connected to every node. In summary, six buffers of 64 arrays of 16 bits and four buffers of 64 arrays of 8 bits are reduced to three buffers of 64 arrays of 16 bits and one buffer of 64 arrays of 8 bits.

The principle of life time analysis in this buffer sharing technique is same as register sharing in high-level synthesis or register allocation in compiler technique. But we apply the principle in the different context which is based on the static schedule of SDF semantic. One difference is that we use the technique not only to same data types but also to different types as well. It is beneficial to modern image processing algorithm like H.264, because they treat blocks of various sizes such as 4×4, 4×8, 8×8, and so forth. In general, VHDL is a strong-typed language so that the sharing of different data types is not simple.

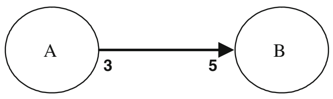


Figure 34. An experimental example for shift-register based buffering.

7. Experiments

We use the example of Fig. 3 in the first experiment. We used Synopsys Design Compiler for synthesis. The clock period is fixed to 20 ns. As a result, one execution of *DCT1D* with propagation delay of 30.2 ns takes two cycles. To evaluate the efficiency of our approach, we compared the quality of the synthesized hardware with a manually designed hardware whose structure is same as Fig. 5a. It includes a *DCT1D* resource and a *transpose* memory of 64×16-bit registers. The controller and the peripherals such as MUXes and counters are also included in the synthesized hardware. The synthesis results of other approaches in Fig. 4 are also compared in Table 2. Since GRAPE tool is not available, we manually implemented the hardware according to the expected synthesis result. So, we admit that there can be some differences between the result of Table 2 and the real GRAPE result. The automatically generated design takes somewhat larger area than the manual design due to conservative buffer allocation and control. In

Table 4. Result of synthesizing Fig. 34.

	Combinational logic (gates)	Sequential logic (gates)	Total (gates)
Write-fixed	137	214	351
Read-fixed	127	204	331
Conservative (TNSE)	169	341	510

our approach, output buffers are allocated separately even when resource sharing is performed. This is the main reason of buffer overuse. Figure 31 displays the trade-off relation between performance and area among various hardware implementations. It also confirms that the proposed technique can explore diverse hardware implementations by simply changing the schedule information.

We also experimented with an example of H.263 decoder in Fig. 23. In this experiment, we compare the automatically generated designs from a dataflow specification (Fig. 23) varying the number of resources. We use only one hardware resource for motion compensation part throughout all experiments, because motion compensation cannot be parallelized beneficially. We use 1, 2, 3, and 6 resources for other blocks to obtain better performance. The VHDL codes for these experiments are automatically generated by simply changing the schedule information. In the synthesized hardware, we assume that the clock period is 20 ns and the memory access takes three clock cycles.

Table 3 shows that the HW area rapidly increases as more HW resources are used. The buffer size is relatively large in this experiment because output buffer for large user-defined type such as 8×8 matrix is implemented using hardware registers. This overhead will be reduced if we use memory type output buffer as frame memory or apply a buffer sharing technique. The results of Table 3 are depicted with an area-delay graph in Fig. 32.

As shown in Tables 2 and 3, we could automatically synthesize various kinds of hardware implementations by simply modifying the schedule information file. And the quality of these implementations is not far from that of manual design so that architecture optimization after automatic generation could be easily performed manually.

Figure 33 show how much the buffer size is reduced by applying the proposed buffer sharing technique. ‘Buffer sharing’ stands for applying buffer sharing only between the same buffer types. ‘Advanced buffer sharing’ denotes the result for applying buffer sharing to all buffer types. The buffer size shrinks to about 44% of the original size by applying the proposed buffer sharing. The glue logic also decreases because the increased overhead due to buffer sharing is less than the decreased control logic associated with the removed buffers.

The last experiment is a toy example of Fig. 34 that demonstrates the usefulness of shift buffering. Table 4 shows the results of synthesizing this SDF graph in three different schemes: *TNSE* scheme, *read-fixed indexing*, and *write-fixed indexing*. To focus on the effect of control logic and buffer, we used dummy blocks for A and B. Table 4 clearly shows the buffer reducing effect. The logic count includes buffers and buffer control logics as well as the centralized controller. The result confirms that the buffer overhead is larger than the centralized controller overhead. You can see that *read-fixed indexing* makes the best result in this example.

8. Conclusion

This paper addresses how to generate RTL code for various hardware structures from a dataflow specification. We proposed a technique to synthesize the hardware architecture by integrating HW library blocks and automatically synthesizing the glue logics and the central controller. By separating the scheduling and HW code generation, we can implement diverse HW architectures from the given dataflow specification by simply changing the schedule and resource sharing information. The proposed technique considers resource sharing and looped schedule to explore wider design space than the previous approaches. Through FRDF specification, more efficient hardware could be synthesized in terms of hardware area. We experimented with two real examples, 2D DCT algorithm and H.263 decoder, to demonstrate how the proposed technique truly builds the working VHDL code, which is verified with a commercial Synopsys tool. For the buffer optimization, we presented two techniques, shift buffering and buffer sharing, to synthesize area efficient hardware by reducing buffers and control logics. We applied the proposed techniques to a real-life example, H.263 decoder subsystem, and obtained significant buffer reduction up to 44%. The synthesized hardware is close to manually optimized one in terms of hardware area. But it is still worse by 7% due to extra buffers between combinational hardware blocks in the synthesized hardware. It remains as a future work to remove the extra buffers. If we can achieve the similar performance as hand-optimized one, we expect that the proposed hardware synthesis methodology can be successfully used in real-life design.

Acknowledgements

This work was supported by Brain Korea 21 project, SystemIC 2010 project funded by Korean MOCIE, and KOSEF research program (R17-2007-086-01001-0). This work was also partly sponsored by ETRI SoC Industry Promotion Center, Human Resource Development Project for IT SoC Architect. The ICT and ISRC at Seoul National University and IDEC provided research facilities for this study.

Note

1. In the original Meyr's work, each node produces one sample on each output arc at each invocation. Therefore, eight invocations are needed to produce eight samples in this example. However, we assume that *DCT1D* block has eight different output arcs in Figure 3 for efficient implementation.

References

1. Synopsis Inc., "COSSAP User's Manual: VHDL Code Generation," 700 E. Middlefield Rd., Mountain View, CA 94043, USA.
2. J.T. Buck, S. Ha, E.A. Lee and D.G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *Int. J Comput Simulation, Special Issues on Simulation Software Development*, vol.4, 1994, pp. 155–182, April.
3. M. Ade, R. Lauwereins and J.A. Peperstraete, "Hardware-software codesign with GRAPE," *IEEE Int. Workshop on Rapid System Prototyping*, 1995, pp. 40–47, June.
4. The CAP Laboratory Seoul National University, "PeaCE Users's Manual v.1.0.1," no. 4, July, 2005. http://peace.snu.ac.kr/research/peace.
5. E.A. Lee and D.G. Messerschmitt, "Synchronous Data Flow," *Proc. IEEE*, vol. 75, no. 9, 1987, pp. 1235–1245, September.
6. H. Oh and S. Ha, "Fractional Rate Dataflow Model for Efficient Code Synthesis," *J. VLSI Signal Process*, vol. 37, 2004, pp. 41–55, May.
7. G. De Micheli, "Synthesis and Optimization of Digital Circuits," McGraw-Hill, 1994.
8. J. Horstmannshoff and H. Meyr, "Optimized System Synthesis of Complex RT Level Building Blocks from Multirate Dataflow Graphs," *Proceedings of the 12th International Symposium on System Synthesis*, 1999, pp. 38–43, Nov.
9. M.C. Williamson and E.A. Lee, "Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph Specifications," in *30th Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, California, USA*, vol. 2, 1996, pp. 1340–1343, November.
10. J. McAllister, R. Woods, R. Walke and D. Reilly, "Multidimensional DSP Core Synthesis for FPGA," *J. VLSI Signal Process*, vol. 43, 2006, pp. 207–221.
11. L. Semeria, K. Sato and G. De Micheli, "Synthesis of Hardware Models in C with Pointers and Complex Data Structures," *IEEE Transactions on VLSI Systems*, vol. 9, 2001, pp. 743–756, December.
12. L. Semeria and G. De Micheli, "Resolution, Optimization, and Encoding of Pointer Variables for the Behavioral Synthesis from C," *IEEE Trans Comput.-Aided Des Integr Circuits Syst*, vol. 20, 2001, pp. 213–233, Feb.
13. N. Vanspauwen, E. Barros, S. Cavalcante and C. Valderrama, "On the Importance, Problems and Solutions of Pointer Synthesis," *15th Symposium on Integrated Circuits and Systems Design*, 2002, pp. 317–322, September.
14. F. Slomka, M. Dorfel and R. Munzenberger, "Generating Mixed Hardware/Software systems from SDL Specifications," *9th International Symposium on Hardware/Software Codesign*, 2001, pp. 116–121, April.
15. O. Bringmann, W. Rosenstiel, A. Muth, G. Farber, F. Slomka and R. Hofmann, "Mixed Abstraction Level Hardware Synthesis from SDL for Rapid Prototyping," *IEEE Int. Workshop on Rapid System Prototyping*, 1999, pp. 114–119, June.
16. P. Zepter, T. Groker and H. Meyr, "Digital receiver design using VHDL generation from data flow graphs," *Proceedings of the 32nd ACM/IEEE conference on Design Automation*, 1995, pp. 228–233.
17. J. Horstmannshoff, T. Grotker and H. Meyr, "Mapping Multirate Dataflow to Complex RT Level Hardware Models," *Proceedings of IEEE Iternational Conference on Application-Specific Systems, Architectures and Processors*, 1997, pp. 283–292, July.
18. J. Hortmannshoff, T. Grotker, H. Meyr, M. Wloka and K. Djigande, "DSP System Synthesis: Integration of Reusable Building Blocks," *Proceedings of international Conference On Signal Processing Applications and Technology*, 1997
19. J. Dalcolmo, R. Lauwereins, M. Ade, "Code Generation of Data Dominated DSP Applications for FPGA Targets," *IEEE Int. Workshop on Rapid System Prototyping*, 1998, pp. 162–167.
20. R. Lauwereins, M. Engels, M. Ade and J.A. Peperstraete, "Grape-II: A System-Level Prototyping Environment for DSP Applications," *IEEE Computer*, vol. 28, 1995, pp. 35–43, Feb.
21. M. Ade, R. Lauwereins and J.A. Peperstraete, "Data Memory Minimisation for Synchronous Data Flow Graphs Emulated on DSP–FPGA Targets," *Proceedings of the 34th conference on design automation*, Anaheim, CA, 1997, pp. 64–69.
22. G. Bilsen, M. Engels, R. Lauwereins and J. Peperstraete, "Cyclo-Static Dataflow," *IEEE Trans. Signal Processing*, vol. 44, no. 2, 1996, pp. 397–408.
23. P. K. Murthy and E. A. Lee, "Multidimensional Synchronous Dataflow," *IEEE Trans. Signal Processing*, vol. 50. no. 8, 2002, pp. 2064–2079.

24. R. Sharp and A. Mycroft, "A Higher-Level Language for Hardware Synthesis," *11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, vol. 2144 of LNCS, 2001.
25. S.S. Bhattacharyya, P.K. Murthy and E.A. Lee, "APGAN and RPMC: Complementary Heuristics for Translating DSP Block Diagrams into Efficient Software Implementations," *DAES*, vol. 2, no. 1, 1997, pp. 33–60, January.
26. W. Sung and S. Ha, "Memory Efficient Software Synthesis Using Mixed Coding Style from Dataflow Graph," *IEEE Transactions on VLSI Systems*, vol. 8, 2000, pp. 522–526, October.
27. H. Jung and S. Ha, "Hardware Synthesis from Coarse Grained Dataflow Specification for Fast HW/SW Cosynthesis," *Proceedings of International Conference on Hardware/Software Codesign and System Synthesis*, Sweden, 2004, pp. 24–29.
28. H. Oh and S. Ha, "Efficient Code Synthesis from Extended Dataflow Graphs for Multimedia Applications," *Proceedings of the 39th Conference on Design Automation*, 2002, pp. 275–280, June.
29. P.K. Murthy and S.S. Bhattacharyya, "Buffer Merging—A Powerful Technique for Reducing Memory Requirements of Synchronous Dataflow Specifications," *ACM Transactions on Design Automation of Electronic Systems*, vol. 9, no. 2, 2004, pp. 212–237, April.
30. H. Oh, N. Dutt, S. Ha, "Shift Buffering Technique for Automatic Code Synthesis from Synchronous Dataflow Graphs," CODES+ISSS'05, Sept. 19–21 2005, pp. 51–56.

Hyunuk Jung received his B.S. and M.S. degrees in Computer Engineering and Ph.D. degree in Electrical Engineering and Computer Science from Seoul National University, Seoul, Korea, in 1998, 2000, and 2005, respectively. He is currently working for Samsung Electronics on system design technology. His research interests include hardware/software codesign, system-level performance analysis, SoC architecture optimization, and embedded system software optimization.

Hoeseok Yang received his B.S. degree in Computer Science and Engineering from Seoul National University, Seoul, Korea, in 2003, where he is currently working towards a Ph.D. degree in Electrical Engineering and Computer Science. His research interests include hardware/software codesign for multi-processor system-on-chip, memory architecture exploration, and hardware synthesis from formal computation model.

Soonhoi Ha received his B.S. and M.S. degrees in Electronics Engineering from Seoul National University, Seoul, Korea, in 1985 and 1987, respectively, and his Ph.D. degree in Electrical Engineering and Computer Science from the University of California, Berkeley, in 1992. He was with Hyundai Electronics Industries Corporation from 1993 to 1994 before he joined as a faculty of the School of Electrical Engineering and Computer Science, Seoul National University, where he is currently a Professor. He is a program co-chair of CODES+ISSS' 2006, ASPDAC' 2008, and ESTIMedia' 2005-6. He has been a member of the technical program committee of several technical conferences including DATE, CODES+ISSS, and ASP-DAC. His primary research interests are various aspects of embedded system design including hardware/software codesign, design methodologies, and embedded software design for multi-processor system-on-chip.