Hardware Synthesis from SPDF Representation for Multimedia Applications

Chanik Park and Soonhoi Ha
Department of Computer Science and Engineering, Seoul National University
Seoul 151-742, KOREA
{park, sha}@iris.snu.ac.kr

Abstract

Even though high-level hardware synthesis from dataflow graphs becomes popular in designing DSP systems, currently used dataflow models are inefficient to deal with emerging multimedia applications since they do not support global parameter update.

In this paper, we propose a VHDL code generation method from synchronous piggybacked dataflow (SPDF) which is an extension of synchronous dataflow (SDF) for representing multimedia applications. Through constructing globally shared memory structure with limited access, we can obtain an efficient RTL architecture in terms of memory and performance compared with other approaches. We demonstrate the usefulness of the proposed approach using a preliminary example of MP3 decoders.

1 Introduction

As multimedia DSP applications proliferate, the need for advanced hardware synthesis tools is increasing to shrink time-to-market window and to obtain performance/cost efficient implementation [1]. In order to reduce the development cycle, system design based on high-level building blocks becomes popular [2].

Today, several hardware synthesis tools generate synthesizable VHDL codes by automating the integration of reusable blocks from synchronous dataflow graph (SDF) specifications [3][4][5]. In an SDF graph shown in figure 1(a), a node represents a function block and an arc or edge between nodes represents data dependency between function blocks. When a node is executed, it consumes a fixed number of samples from each input arc, and produces a fixed number of samples to each output arc. The numbers on the edges denote the number of samples produced or consumed at each port. The small diamond symbol on an arc indicates an initial data. Since the SDF is suitable to describe multi-rate DSP algorithms and fully analyzable at compile-time, the SDF has been preferred in prototyping DSP systems which require high performance. In case of a coarse grain dataflow graph, a function block may contain an algorithmic state or parameter1 (denoted with a gray box on top of a node in figure 1(a)). The state or parameter is not accessible from other nodes in the SDF semantics to avoid side effects.

To generate synthesizable VHDL code from a given SDF graph, each function block is mapped to a corresponding hardware component from predefined block library. And, to satisfy the dataflow semantics, additional interface logics between blocks and control logics should be synthesized with RTL components [6]. Figure 1(b) shows the RTL level implementation of figure 1(a) on target architecture. In order to implement internal states or parameters, registers or memory can be used. Since the additional interfaces control logics may have serious effect on the system efficiency in terms of cost and performance, many researches have focused on the interface problem [3][4][5].

Figure 1: RTL architecture generation from an SDF graph

Even though the hardware synthesis from SDF has been more or less successful, we often meet with some difficulties in designing multimedia applications with existing reusable blocks. In multimedia applications, data is compressed into bit streams to minimize transmission cost and storage requirement. The MPEG standard defines the format of bit streams for de-multiplexing compressed audio and video data into original signals [7]. Figure 2 shows the MPEG2 video stream structure and a possible architecture for an MPEG2 video decoder. Usually, the format consists of a node’s executions, while the parameter determines a specific behavior of a node from the more generic definition of the node. In this paper, we do not distinguish them though our main focus lies in the parameter.

1 Rigorously speaking, local state and parameter are distinguished. The local state saves some information between
nizing it with 100 data samples. The integration becomes problematic since the rate consistency of the graph [9] and efficient implementation should be considered.

![Figure 3: Problematic integration](image)

Even though there is no clean solution in a pure SDF representation, several approaches are given in figure 4. The first possible approach is to duplicate the parameter 25 times (figure 4 (a)); here, the number, 25, is computed in order to maintain the rate consistency. However, this approach requires extra buffers on the arc between DEMUX and GAIN. The second approach is to group the data samples as shown in figure 4 (b). However, this not only requires large data buffers between the FIR block and the GAIN block, but also removes the possibility of parallel processing of 25 data samples, which results in reducing the possible maximum performance. Another factor that needs attention is the reusability of the GAIN block. Surely, it is an undesirable burden to designers.

![Figure 4: Dataflow solutions for synchronous state update request](image)

Cyclo-static dataflow model [10] gives an improved solution as shown figure 4 (c). In cyclo-static dataflow model, the number of samples produced or consumed per node execution is not fixed. Instead, it follows a cyclic pattern. At the first firing of node DEMUX, one global parameter is generated while the next 99 firings do not generate any sample. By allowing such a cyclic pattern, cyclo-static dataflow model solves buffer requirement problem. However, it does not solve the reusability problem. Another GAIN block with an extra port should be designed to accept parameter value from the outside.

These three solutions may suffer from the synchronization between data samples and parameters when the data path from the DEMUX block to the GAIN block has pipeline delays or sample rate changes. For example, if there are initial samples (or pipeline delays) between the FIR block and the GAIN block, there should be the same amount of
initial samples on the parameter arc between the DEMUX block and the GAIN block. Otherwise, synchronization between data samples and parameter will be broken at the execution of the GAIN block. Consequently, the path for the parameter update should be consistent with that for data samples.

Another inefficiency comes from the fact that they do not allow sharing the control parameters between blocks. Suppose the "gain" parameter is needed in another downstream actor. All these solutions need a separate arc to deliver the parameter to the actor using a local storage.

Even though several approaches of generating hardware from synchronous dataflow graphs are known to date [3][4][5], the limitation caused by the specification model leads to inefficient or incorrect implementations.

Synchronous piggybacked dataflow [11] is presented in figure 4 (d) where the global parameters are allowed to periodically update the local parameters of the blocks. However, the implementation has been restricted to single processor target. In this paper, we propose a VHDL code generation approach based on the work discussed in [11].

3 Input Specification - SPDF

In [11], we introduced an extension of SDF to add the notion of global parameters to SDF graph without any side effect. The key idea is to piggyback the global parameters on each data sample. Such piggybacking idea is realized with a global shared memory structure and its restricted access control. This simple extension solves the buffer requirement, reusability and synchronization problems. Figure 5 shows the corresponding SPDF graph represented with the blocks in figure 3.

To manage a parameter of a block as a global parameter, we define a global memory structure. The global memory maintains the outstanding values of the parameters updated from the outside. It allows only one writer and many readers. Special blocks called piggybacking block (PB) and state convert (SC) are introduced to piggyback the parameter on the associated data samples. The SC block consumes a parameter value from the preceding block and produces parameters as many as the number of related data samples. The PB accepts a parameter value from the SC block and a data sample from the preceding block, updates the global memory structure with the received parameter value. However, differently from data port, the output port of the SC block delivers only one sample followed by null samples, thus requires only one buffer on its output arc.

A global parameter is periodic if it is periodically updated after a fixed number of samples, while it is aperiodic if it is updated at irregular intervals. In this paper, we focus on periodic parameter update since the aperiodic parameter update can be emulated with periodic update with period of 1. In order to synchronize the parameter update with the relevant data samples, we define a property pair \([\text{period}, \text{offset}]\) for both the PB block and the blocks with global parameters. The period is the repetition period of updating a global parameter, and the offset is the starting offset of updating the parameter, both in terms of node execution. We denote the property pair of node \(n\) for a global state \(g_s\), as \(\{\text{period}_{n,g_s}, \text{offset}_{n,g_s}\}\).

For example, suppose that the global parameter named "gain" changes every 100 data samples and an initial sample (diamond symbol) is placed on the arc between the FIR block and the GAIN block in figure 5. The PB block has the period of 100 and the offset of 0. This means the PB block updates the "gain" parameter every 100 executions. On the other hand, the GAIN block has the period of 25 and the offset of 1. Then, the GAIN block updates its local parameter after its second execution every 25 executions. The change of period and offset reflects the path information between the source block and the target blocks of the parameter. The \(\{\text{period}_{PB,g_s}, \text{offset}_{PB,g_s}\}\) of the PB block is manually supplied by a user, while that of each target block is automatically computed at compile-time. Note that the size of the global memory structure for the "gain" parameter is determined to be 2. The memory requirement of global parameters is decided through the static analysis of the SPDF graph at compile-time.

Figure 5: An illustration of the SPDF graph

4 Static Analysis of SPDF Graph

In this section, we present a static analysis of an SPDF graph to determine the \(\{\text{period}, \text{offset}\}\) of the PB block and the target blocks.

An SDF model has an advantage that the graph syntax is fully analyzable at compile-time [9]. Since an SDF graph imposes only partial ordering constraints between the nodes, we can exploit parallelism and determine the firing order of nodes in various ways to meet the design objectives; this is called a schedule. A valid schedule is a finite schedule that does not deadlock and produces no net change in the number of tokens accumulated on each arc. The repetition count, \(q_j\) defines the total number of firings of a node \(j\) in a valid schedule.
An SPDF graph \( G \) consists of an ordered pair \((N, E)\), where \(N\) is a set of nodes and \(E\) is a set of edges. A state-path is a simple path through which a global state is propagated from a PB block to a target node. If a node on a state-path consumes and produces data samples piggybacked with the same value of a global state, we call the node state-consistent on the state-path. Consider two adjacent nodes \( A \) and \( B \) on a state-path as shown in figure 6. If node \( A \) is state-consistent, for node \( B \) to be state-consistent, the following equations should be satisfied.

\[
\begin{align*}
\text{period}_{A, B} &= \text{period}_{A, B} \times q_n \frac{q_n}{q_A} \\
\text{offset}_{A, B} &= \text{offset}_{A, B} \times O_A + D_{A, B} \\
\end{align*}
\]

(1)

\[
\begin{align*}
\{\text{period}_{A, B}, \text{offset}_{A, B}\} = \{\text{period}_{A, B}, \text{offset}_{A, B}\} \\
\end{align*}
\]

Figure 6: Adjacent nodes on a state-path

In order to determine the state-consistency of a node on a state-path, we start with a PB block and traverse the nodes on the state-path computing \{period, offset\} of nodes until we reach the end of a state-path or a state-inconsistent node.

For a node on a state-path of SPDF graph \( G \), we call the node state-consistent on the state-path, if the consumed and the produced samples for each firing on the state-path are all piggybacked with the same value of a global parameter. Table 1 shows the \{period, offset\} of nodes on a state-consistent path in figure 5 according to the equations (1) and (2).

Table 1: \{period, offset\} of nodes on a state-consistent path in figure 5

<table>
<thead>
<tr>
<th>(period, offset)</th>
<th>PB</th>
<th>FIR</th>
<th>GAIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>{100, 0}</td>
<td>{25, 0}</td>
<td>{25, 1}</td>
<td></td>
</tr>
</tbody>
</table>

5 SPDF-VHDL Implementation

When we implement an SPDF graph with multi-rate nodes on a hardware target, we can consider two schemes. One is to take the sequential execution based on multiple clock triggering scheme [3][4] (figure 7(a)). Another is to execute the nodes in parallel as much as possible [5], which is our current scheme (figure 7(b)). Since there is trade-off between these two schemes in terms of performance and cost, the hybrid approach to compromise them remains as a future work (figure 7(c)). In this paper, we assume that the target architecture for generating VHDL code from the SPDF graph is based on either the parallel execution or the sequential execution.

Once the scheme is selected, the number of instances for each node \( n \) (we denote it as \( \text{Instances}(n) \)) is determined. Thus, the \{period, offset\} pairs should be adjusted according to the number of outstanding instances for PB blocks and target blocks as shown in equations (3) and (4).

\[
\begin{align*}
\text{adjusted - period}_{A, B} &= \frac{\text{period}_{A, B}}{\text{Instances}(n)} \\
\text{adjusted - offset}_{A, B} &= \frac{\text{offset}_{A, B}}{\text{Instances}(n)} \\
\end{align*}
\]

(3)

(4)

Figure 8 shows the overall algorithm for generating memory structure and control logic to handle global parameters. In Step 1, we examine all PB blocks and make a list. In Step 2, we check if each PB block has a valid period of a global parameter. In Step 3, we identify target blocks with global parameters and make a list of target blocks for each global parameter. From Step 1 to Step 3, we traverse the given graph to gather information for PB blocks and the target blocks.

```
Constructing global memory structure and control {
    Step 1: examine the piggybacking blocks
    Step 2: checking the periods of global parameters
    Step 3: identify the blocks with global parameters for each piggybacking block
    Step 4-1: determine \{period, offset\} pairs of the target blocks
    Step 4-2: construct the optimal buffer sizes
    Step 4-3: attach update control logic to the target blocks
}
```

Figure 8: Algorithm for constructing global memory structure and control
In Step 4, the \{period, offset\} pairs for target blocks are determined. For each global parameter, the optimal size of the global parameter buffer is determined. After Step 4-1 and 4-2 are computed, our framework glues control logic to the target block. The control logic plays a role of copying a global parameter into the local parameter. Figure 9 shows the automatically generated RTL architecture for handling a global parameter.

In figure 9, the update control logic is incorporated into the centralized controller\(^2\) for the blocks with global parameters. Control logic for managing a global parameter is automatically generated based on the properties of the PB block and the target blocks. For example, as shown in figure 10, assume that the PB block and the target block are executed every clock; in fact the PB block has no execution body but gives the control information such as “period” and “offset” to the central controller. Then, at the reset time, “offset” values (P_o and T_o) are loaded into PB counter and target counter, respectively. When the PB block is executed “offset” times, “write_enable” signal is activated and a new parameter is loaded into parameter FIFO. On the other hand, when the target block is executed “offset” times, “read_enable” signal is activated to load the current global parameter value into the target blocks. At the same time, the “write_enable” and “read_enable” signals enable the “period” values (P_p and T_p) to be loaded into each counter. Then, periodic parameter update is executed according to this counter-based control. Figure 11 shows RTL architectures from the SPDF graph of figure 5. Since the “offset” value of the GAIN block is 1, a buffer is inserted before the first instance of the block, while other instances share a parameter buffer without any local buffer (figure 11(a)). Note that we can hardly obtain such implementation of sharing a parameter from the models such as SDF and CSDF that are mentioned in section 2. In case of sequential execution of figure 11 (b), just one buffer is located differently from the duplicated buffers of figure 4 (a).

6 A Preliminary Experiment

We have implemented the proposed scheme in our developing PeaCE environment [12], which is an extension to Ptolemy [13] as codesign environment.

---

\(^2\) There are two approaches, distributed [4] and centralized [3] control approaches, to satisfy the correct behavior of dataflow graph. Our framework is based on centralized control approach.
We applied our approach to a real MP3 decoding algorithm whose implementation is based on the MPEG standard [7]. Figure 12 shows an SPDF representation of MP3 decoder. The bottom of figure 12 shows the inside graph of dequantizing block (DQ). We express the internal of the DQ block, permitting exploitation of parallelism within the DQ block.

To compare the implementation overhead between the SDF and the SPDF, we synthesized the generated VHDL code for the DQ block on the FPGA target [14]. The large rate change between DEMUX block and DQ block made memory overhead more outstanding compared with other factors such as control and computation logic. Table 2 shows the comparison in the approximate number of logic gates. Integration overhead includes buffer memory and control logic to implement global parameter update. In case of the SDF (grouping), data grouping to satisfy rate consistency causes large buffer requirement. On the other hand, the SPDF has more control logic overhead than the SDF. However, this is trivial overhead considering the reduction of memory requirement. We excluded the SDF (duplication) approach from the comparison, since the buffer requirement for parameter arc exceeds that for data buffer. Although the CSDF approach, in terms of implementation overhead, is comparable to the SPDF, the reusability and synchronization problems still happen.

Table 2: Comparison of synthesized results

<table>
<thead>
<tr>
<th>Model</th>
<th>Area(gates)</th>
<th>Memory(gates)</th>
<th>Integration overhead(gates)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDF(grouping)</td>
<td>54084</td>
<td>80416</td>
<td>8280</td>
</tr>
<tr>
<td>SPDF</td>
<td>54084</td>
<td>80416</td>
<td>9480</td>
</tr>
</tbody>
</table>

7 Conclusions

Though there have been some approaches to generate VHDL code from the SDF graphs, their implementations are inefficient to deal with emerging multimedia application since they lack in global parameter handling. This inefficiency results from the limitation of the specification model.

In this paper, we implemented a VHDL code generation method from the SPDF representation for multimedia application. In our framework, the global memory structure and control logics are automatically generated. With this feature, we can easily integrate the existing blocks without redesign. The simple experiment with a part of MP3 decoder showed the efficiency accomplished with our approach. Though we used FIFO memory for implementing the global data structure, we can consider random access memory as a future work.

8 Acknowledgement

This research was supported in part by the Korean Ministry of Information and Communication (MIC) under the program of University Research.

References

[7] ISO/IECTC1/SC2/WG11 MPEG 91/0.., Nov. 1991, Committee Draft “Coding of moving pictures and Associated audio for digital storage media at up to about 1.5Mbit/s”.