Dataflow Specification for System Level Synthesis of 3D Graphics Applications

Chanik Park, Sungchan Kim and Soonhoi Ha

Computer Science and Engineering
Seoul National University
Seoul, 151-742, KOREA
Tel : +82-2-880-7292
Fax : +82-2-879-1532
e-mail : {park, ynwie, sha}@iris.snu.ac.kr

Abstract - 3D graphics is becoming an important application area together with multimedia applications. The dynamic behavior of 3D graphics application brings new challenges for system level specification and synthesis methodologies. Although existing dataflow models are successfully used for DSP system design, they are not sufficient to deal with 3D graphics algorithms since they lack in global state management and dynamic behavior handling. In this paper, we propose an extended synchronous piggybacked dataflow model with dynamic constructs for representing 3D graphics algorithm. We also present implementation techniques for software and hardware synthesis from the specification. With a simple 3D graphics pipeline, we show the novelty and usefulness of the proposed specification model.

I. Introduction

As the increase of design complexity in designing multimedia and 3D graphics applications is expected to outgrow the design power with the existent design methodology in the near future, the system-level design methodology gains significant research activities [1]. As of now, the most successful system-level design tools can be found in the DSP system design [2][3][4]. The main reason, we believe, is that there is a well-known formal specification model that is synchronous dataflow (SDF) model or its variant, for DSP applications. The formal properties (e.g. provable liveness and bounded memory) of SDF model ease the task of system verification as well as automatic system synthesis. The importance of formal specification will be greater as the system validation becomes more difficult along with the increased system complexity.

Although the system-level design methodologies gain significant research activities in the DSP system design, very little research has been done in the 3D graphics area. Since multimedia and 3D graphics applications will be key applications in the future digital system, we need more general specification model. These applications require either the use of global states or dynamic constructs such as conditionals and data-dependent iterations. Recently, synchronous piggybacked dataflow (SPDF) model extended the SDF model with controlled global states to satisfy the former requirement, not the latter. On the other hand, any specification model for the 3D graphics applications needs to represent the system's dynamic behavior. Moreover, the specification model should have capability of automatic software and hardware synthesis, to enable the system level design.

The goal of this work is to develop an efficient specification method for 3D graphics and implement automatic SW/HW synthesis from the specification. In this paper, we base our representation on SPDF graph representation.

The rest of the paper is organized as follows. In section II, we characterize the basic polygonal 3D graphics pipeline and discuss related work in literature. In section III, the proposed specification method is explained. Section IV explains implementation techniques to synthesize software and hardware system from the proposed specification. Finally, we show some experimental results using a simple 3D graphics pipeline and make conclusions in sections V and VI.

II. Background

A. 3D Graphics Pipeline

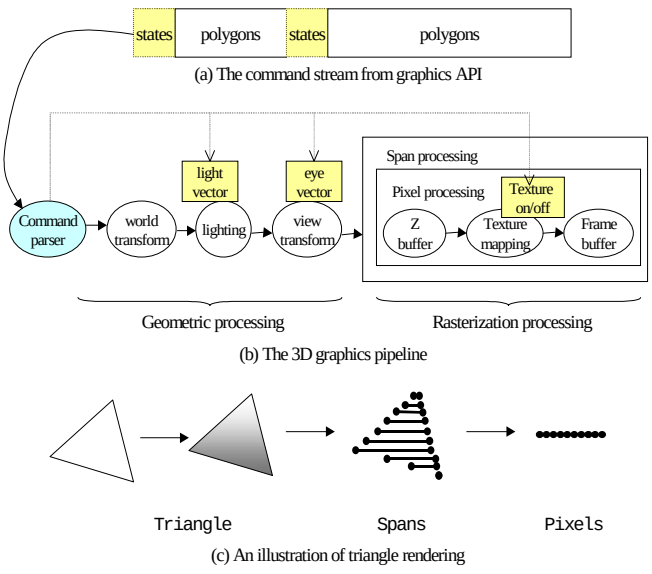


Fig. 1. Illustration of 3D graphics pipeline

A 3D graphics application generates a series of frames. Each frame consists of a collection of 3D objects that are represented with polygons (usually triangles) and an associ-

ated set of characteristics, such as light, color, shade, texture, etc. In case of interactive applications such as animation and games, the number of polygons in a frame and the set of characteristics tend to be changed dynamically.

Fig. 1 shows a typical 3D graphics rendering pipeline. The input to the pipeline is the command stream from 3D graphics application programming interface (API) such as OpenGL and DirectX [6][7]. The first block, `Command parser`, interprets the command streams, delivers the triangles to downstream blocks of the pipeline and sets the local state variables of associated blocks. The states are associated with the following polygons until new states are set. Normally, these states are shared by many polygons and their updates are aperiodic[1]. Thus, it is natural to implement these states using shared global states. The `world transform` block converts the vertices from object space to world space, for all objects within the frame boundary. The `lighting` block computes the effects of light sources. The `view transform` block converts the vertices from world space to camera space, with the eye vector. The `rasterization` block converts the transformed triangle into pixel values and stores them in frame buffer for display. The `rasterization` block itself is comprised of two steps. At the first stage, a triangle is decomposed into multiple spans which are denoted as lines with rounded ends as shown in Fig. 1 (c). These spans are again decomposed into a set of pixels. In order to determine the visible pixels, depth test is executed in `Z-buffer` block. Optionally `texture mapping` is applied to the pixel depending on the `on/off` state. Finally the rendered pixel is written into `frame buffer`. As shown in Fig. 1 (c), the number of spans and pixels to be rendered is dependent on the size of the triangle and determined at run-time.

As explained above, 3D graphics algorithm characterized by the following features:

- **More data-driven than control driven**
- **Data-dependent iteration**
- **Aperiodic state update**

Thus, a specification for 3D graphics system is required to represent the above features efficiently since a good implementation starting from a poor representation is hardly expected.

B. Related Work

In [9], the authors proposed the algorithm prototyping environment (APE) based on C++/VHDL. Although the APE provides the function of exploring diverse graphics algorithm with reusable C++ classes, they do not support automatic SW/HW synthesis from the specification nor static analysis of the specification.

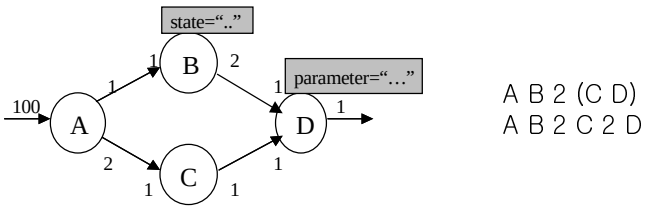
In DSP community, the SDF-related models have been preferred since their semantics is well matched with the DSP algorithms and their formality enables fast prototyping of DSP systems [4][5][8]. However, the SDF-related models are not appropriate to represent 3D graphics algorithm since the aperiodic state update from the outside block and the data-dependant iteration of the algorithm cannot be represented.

In [10], software synthesis for dynamic dataflow (DDF) graph that supports the data-dependent iteration has been introduced. However, it suffers from runtime overhead if it is implemented into a real software or hardware system.

The synchronous piggybacked dataflow (SPDF) graph is proposed to represent multimedia applications with global states such as MP3 decoder in [12]. Though the SPDF gives an improved solution to handle periodic or aperiodic global states, 3D graphics applications are still out of reach since there is no method to specify the data dependent iteration in the SPDF.

To our best knowledge, there is no system level specification method to represent 3D graphics algorithm which requires both data-dependent iteration and aperiodic global state update. In this paper, we propose an extended SPDF model for expressing the 3D graphics algorithm and present efficient hardware/software implementation techniques from the proposed model.

III. Synchronous Piggybacked Dataflow Model with Dynamic Constructs



(a) An example of an SDF graph (b) Some schedules

Fig. 2. An SDF graph and some schedules

Fig. 2(a) represents an example of an SDF graph. Each circle represents a node, or a function block ranging from fine-grain operation such as addition or multiplication to coarse-grain subsystem such as Discrete Cosine Transform (DCT) or Texture mapping. When a node is executed, it consumes a fixed number of data samples from each input arc and produces a fixed number of data samples to each output arc. In Fig. 2(a), each arc is annotated with the number of data samples produced or consumed by its source or sink. A coarse grain block may contain states inside, called *local states* or *parameters*, to save internal information between successive executions of the block. The gray boxes above nodes denote the local state and parameter that belong to the nodes. The SDF has advantage that it is statically ana-

[1] A state is periodic if it is periodically updated after a fixed number of data samples, while it is aperiodic if it is updated at irregular intervals.

lyzable at compile-time and parallelism can be extracted automatically from the representation. For implementation of an SDF graph, the execution order of nodes is determined to meet the design objectives; this is called a schedule. Fig. 2 (b) shows some possible schedules of Fig. 2 (a).

In [12], the SPDF model is introduced to add the notion of global states to SDF graph without any side effect. The key idea is to piggyback the global states on each data sample. Such piggybacking idea is realized with a global memory structure and its restricted access control. Fig. 3 shows the corresponding SPDF graph for the geometric processing.

To manage a state of a block as a global state, we define a global state table (GST). The GST maintains the outstanding values of the states updated from the outside. It allows only one writer but possibly many readers. Special blocks called piggybacking block (PB) and state convert (SC) are introduced to piggyback the state on the associated data samples. The SC block consumes a state value from the preceding block and produces states as many as the number of related data samples. The PB accepts a state value from the SC block and a data sample from the preceding block, updates the GST with the received state value.

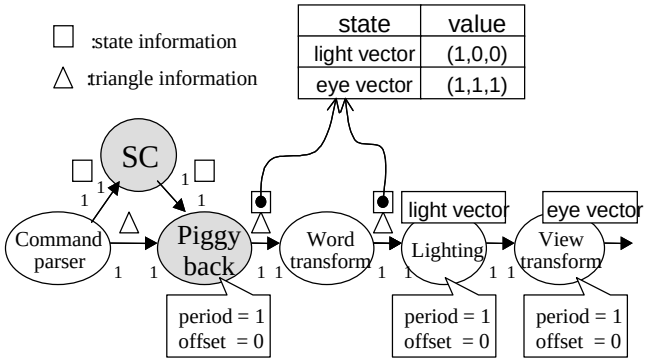


Fig. 3. An SPDF graph for geometric processing

In order to synchronize the state update with the relevant data samples, we define a property pair {*period, offset*} for both the PB block and the target blocks with global states. The period is the repetition period of updating a global state, and the offset is the starting offset of updating the state, both in terms of node execution. The {*period, offset*} of the PB block is manually supplied by a user, while that of each target block is automatically computed at compile-time.

For example, suppose that the global states named “light vector” and “eye vector” change aperiodically in Fig. 3. Then the PB block is supplied with the period of 1 and the offset of 0. This means the PB block updates the “light vector” and “eye vector” states every execution. On the other hand, the Lighting and View transform blocks have the period of 1 and the offset of 0. Then, the blocks update their local states every execution.

Note that the size of the GST entry for the “light” and “eye” states is determined to be 1. The memory requirement of global states is decided through the static analysis of the SPDF graph at compile-time.

On the other hand, the SPDF model cannot express data-dependent iterations as shown in rasterization processing of Fig. 1. As illustrated in section II, the number of pixels to be processed in one frame depends on the size and the number of triangles known only at run-time.

Now, we define dynamic constructs to allow data-dependent iteration and conditional execution of the SPDF graph. Some constructs are shown in Fig. 4 [11]. In Fig. 4 (a), the SWITCH node consumes one data sample from the input arc and one Boolean sample from the control arc, C. Depending on the Boolean value received from the control arc, it produces one output sample either to the true arc, T, or to the false arc, F.

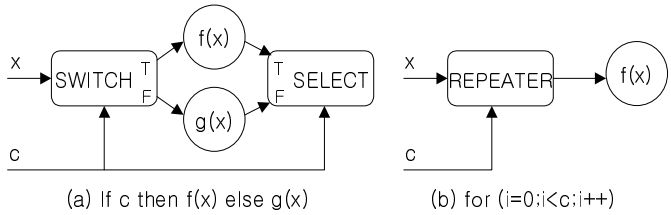


Fig. 4. Dynamic constructs

In Fig. 4 (b), the number of data samples produced on the output arc of the REPEATER is determined by the control value, C, hence it is data-dependent. The f(x) is executed as many times as the number indicated by the control value, C. Though the dynamic constructs remove the advantage of static scheduling, the run-time scheduling overhead can be minimized by quasi-static scheduling [13].

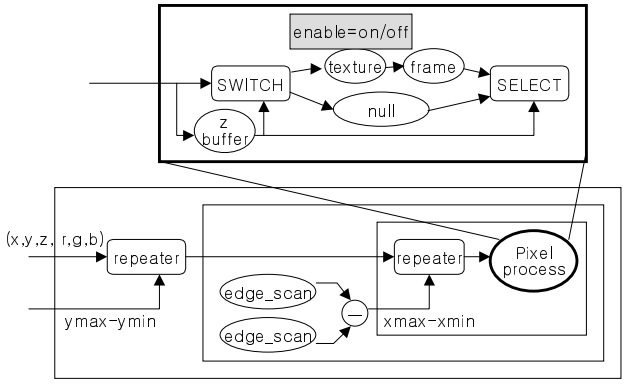


Fig. 5. An SPDF representation for rasterization

Fig. 5 represents an SPDF graph with “for-loop” and “if-then-else” constructs for rasterization processing. In the outer loop, the number of spans (ymax-ymin) is provided as a control value and the number of pixels is given to the inner loop as a control value. In `Pixelprocess` block, Z buffer tests depth of pixel and the comparison result determines whether the pixel continue to be processed in the texture and frame blocks or not. The local state named “enable” in `texture` block is registered as a global state and controlled by another block; in this case `Command parser` is a writer of the global state.

In this section, we added dynamic constructs to the SPDF model and incorporated the global states into the dynamic

constructs seamlessly. Through this extended model, we can represent the 3D graphics algorithm which has data-dependent iterations as well as global states. In the next section, we present how the SPDF specification is compiled into C code and VHDL code for software and hardware synthesis, respectively.

IV. System Synthesis from the Extended SPDF Graph

A. *Code synthesis for global state update*

In this section, we present a technique for efficient SW/HW implementation of the extended SPDF graph. We start from a given schedule; scheduling SDF with dynamic constructs can be referred to [11]. To generate C/C++ code for software and VHDL code for hardware, each node in a schedule is mapped to a corresponding component from pre-defined library blocks which are hand-optimized or provided by IP vendors. Fig. 6 (a) shows the scheduling result generated from Fig. 3. Note that the state update (SU) code is automatically augmented in our framework before the blocks `lighting` and `view transform` since they contain global states. Fig. 6 (b) and (c) show parts of the synthesized code from Fig. 6 (a). The PB block does nothing but passing the data samples between the previous block and the next until the internal count is equal to the offset. The `lighting` and `view transform` blocks will work as usual without awareness of the change of their states since the preamble codes (SU) for their states update are inserted at compile-time as shown in Fig. 6 (c). The GST logically corresponds to a circular buffer, so it is implemented with a fixed size of memory and a write pointer and multiple read pointers. If the size of a GST is one, the SU code is omitted. Instead, the PB block directly updates the local states of the target blocks.

Command parser **SC** **PB** world transform **SU** lighting **SU** view transform

(a) Scheduling result from Fig. 3

```
if (count==offset){
  new_value=read_state_port();
  update_global_table
     ("light vector",new_value);
}
data_out[]=data_in[];
If (++count >= period) count=0;
```

(b) PB code

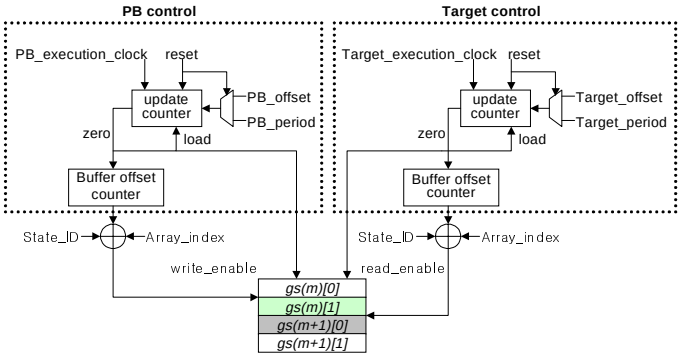
```
{// SU request code
 if (count==offset)
    light_vector=read_global_table
              ("light vector");
 if (++count >= period) count=0;
}
Normal code for Lighting
…
```

(c) SU and the lighting code

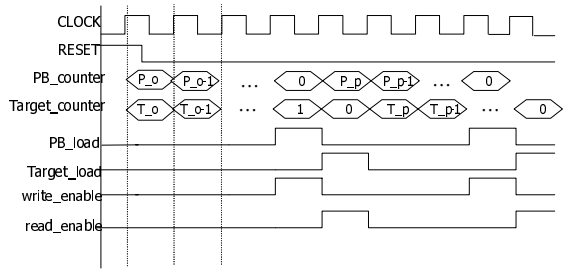
Fig. 6. Pseudo synthesized code

As for hardware synthesis, the GST can be implemented with any kind of memory such as one-port RAM, dual-port RAM or FIFO depending on give design constraints (Fig. 7 (a)). In case shared bus is used for accessing the GST, an arbitration logic may be necessary. Regardless of a memory architecture on which global memory is implemented, the general control logic for PB and target blocks should be automatically generated.

The general control logic for PB and target blocks is designed to be a counter-based control logic as shown in Fig. 7(a). In fact, the PB block has no execution body but gives the control information such as "period" and "offset" to the control logic at synthesis time. For example, assume that the PB block and the target block are executed every clock as shown in Fig. 7(b). Then, at the reset time, offset values (P_o and T_o) are loaded into the PB counter and the target counter, respectively. When the PB block is executed offset times, "write_enable" signal is activated and a new state is loaded into the GST memory. On the other hand, when the target block is executed "offset" times, "read_enable" signal is activated to load the current global state value into the target blocks. At the same time, the signal enables the period values (P_p and T_p) to be loaded into each counter. Then, periodic state update is executed according to this counter-based control. In case of aperiodic state update, the period values are determined to be all 1s. Thus, the global state is updated every execution.



(a)



(b)

Fig. 7. Control logic (a) and timing diagram (b) for managing a global state

An address to access the GST memory consists of three fields as shown in Fig. 7(a). The first field, *State_ID*, identifies a unique state in the GST. The second field, *Buffer_offset*, indicates a currently referenced state among the states with *State_ID*. The third field, *Array_index*, is necessary to refer to one of an array if a state is an array such as a texture map.

B. *Code synthesis for dynamic constructs*

Synthesized C code from the specification for rasterization (Fig. 5) is shown in Fig. 8. For detailed scheduling method for dynamic construct, [11] can be referred. In the outer loop, the loop bound is determined by the number of spans

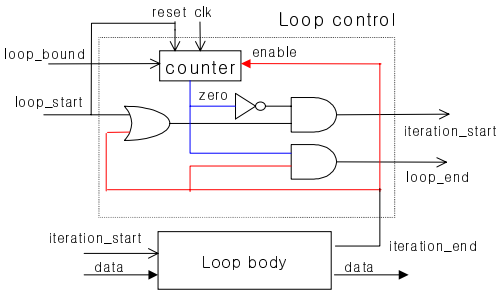
spans in a triangle at run-time, while the inner loop bound is computed depending on the number of pixels on spans.

```
{ int i;
for (i = 0; i < output_51; i++){
 {  /* star Repeater1_36 */
      output_270 = output_1_53;
 }
 {  /* star 3Drender.Rasterize1.BRamp6  */
 }
                              ....
 { int i;
 for (i = 0; i < output_285; i++) {
    {  /* star Repeater1_72 (class CGCRepeater) */
       output_345 = output_1_286;
    }
    {  /* star 3Drender      Z compare
      if (z31[addr]>old_z32[addr]) {
        flag_353=0;
      }
      else flag_353=1;
    }
    switch((int)flag_353) {
      case 0 : {
                  if (g_texture_enable) // global state
                          texture mapping;
                  frame write ;
                  break;}
      case 1: { break; }
    }
 }}}}
```

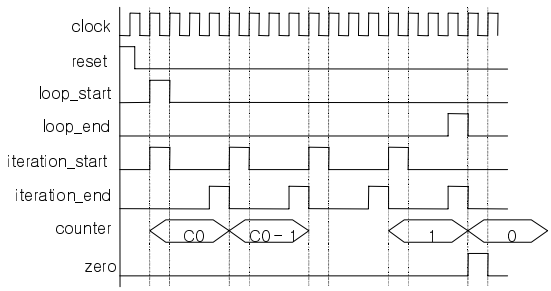
Fig. 8. Synthesized software of dynamic constructs

In order to implement the conditional execution of the nodes, “switch-case” construct is generated. The flag set by the `Z-buffer` block decides the execution path for the pixel. The `SU` code for handling the global state named “g_texture_enable” is automatically augmented before texture mapping block.

For hardware implementation, a structural VHDL code mapped to each block is generated and control logic to handle the semantics of dynamic constructs is glued onto the synthesized code. For example, the hardware implementation of “for-loop” construct is shown in Fig. 9. A loop is activated by “loop_start signal” which indicates all data are available for the execution. At the same time, the “loop_bound” value is loaded into counter and “iteration_start” signal triggers the loop body for an iteration. When an iteration is finished, the “iteration_end” enables the counter to decrease the counter value by one. Finally, when the counter value becomes zero, the “loop_end” signal is delivered to the upper loop or next block.



(a)



(b)

Fig. 9. Control logic (a) and timing diagram for For-loop construct

V. Experimental Results

We have implemented the proposed scheme in our developing PeaCE environment [14], which is an extension to Ptolemy [15] as a codesign evironment.

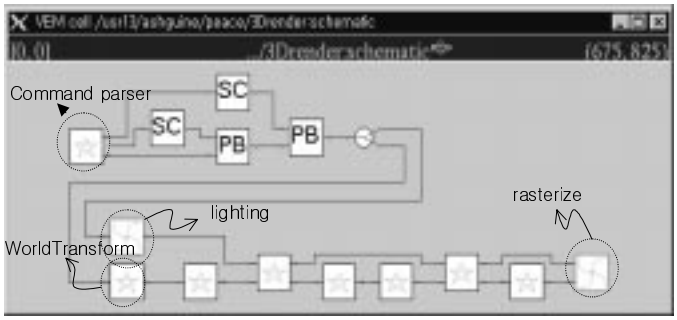


Fig. 10. Top level representation of 3D pipeline

Fig. 10 shows top level representation of 3D pipeline. The `Lighting` block and `WorldTransform` block are executed in parallel. The `rasterize` block is a hierarchical block that has the nested loop as shown in Fig. 5. After generate a C code and a VHDL code for Fig. 10, we perform two experiments to verify the synthesized codes from our specification, In the first experiment, the 3D pipeline is executed only on the ARM7TDMI processor and in the second experiment, the pipeline is partitioned manually into geometry (SW) and rasterization (HW) parts. The hardware code is targeted for ALTERA FLEX10K [19]. Z-buffer and frame buffer are allocated on two SRAMs, repectively as shown in Fig. 11.

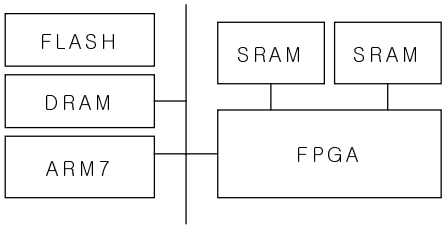


Fig. 11. Target architecture

Table 1 shows the comparison between execution cycles measured in two experiments when 1000 triangles are rendered through the simple 3D pipeline. As expected, the software 3D pipeline takes longer time than the software and

hardware 3D pipeline since the operations and the accesses to Z-buffer and frame buffer are serialized in the software only implementation, while the hardware implementation can exploit more parallelism than the software implementation.

Table I. Execution time comparison

	software only	software and hardware
exectution cycles	16141234	5590113

In this experimentation, we measured the hardware and software execution times by simulation tools such as ARMulator [16] and Maxplus-II [17]. The execution cycles were computed assuming that ARM7TDMI processor and FLEX10K share the same system clock. Also the interface logic was hand-written before simulation. As of this writing, we are implementing the prototyping environment to obtain the real execution time data.

VI. Conclusions

We presented a formal specification model for 3D graphics pipeline and implemented C and VHDL code synthesis from the specification. Our proposed model added the feature of dynamic constructs to the SPDF model, which is essential to represent the dynamic behavior of 3D graphics pipeline.

There is no formal model known to us except the proposed one to represent the 3D graphics applications for system-level synthesis. We verified that the proposed model is automatically synthesizable to software and hardware with an example of the conventional 3D rendering pipeline.

As a future work, automatic partitioning of the extended SPDF graphs and exploiting data parallelism in data-dependent iteration remain to be considered. In order to prove the efficiency of automatically generated code, comparison with manual implementation should be made.

Acknowledgements

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

References

[1] P. Lippens, V. Nagasamy, W. Wolf, CAD Challenges in Multimedia Computing, ICCAD, pp. 502-508, 1995.

[2] R. Lauwereins, Marc Engels, Marleen Ade, and J. A. Peperstraete. Grape-II: A system-level prototyping environment for DSP applications. IEEE Computer, pp. 35-43, Feb, 1995.

[3] Synopsis Inc., 700E. Middlefield Rd., Mountain View, CA 94043, USA, COSSAP User's Manual : VHDL Code Generation.

[4] P. Zepter and T. Groker and H. Meyr, Digital receiver design using VHDL generation from data flow graphs, In Proc. 32th Design Automation Conf., June 1995.

[5] E. A. Lee and D. G. Messerschmitt, Synchronous data flow, Proc. of the IEEE, Sept, 1987.

[6] Jackie Neider, Mason Woo, and Tom Davis. *OpenGL Programming Guide*. Addison-Wesley, Reading, Ma., 1993.

[7] Microsoft. *Direct3D Immediate Mode*. From the DirectX 6.1 SDK.

[8] Greet Bilsen, Marc Engels, Rudy Lauwereins, J.A. Peperstraete. Cyclo-Static Dataflow. IEEE Transaction on Signal Processing, Vol. 44, No. 2, Feb. 1996.

[9] Jon P. Ewins, Phil L. Watten, Martin White, Michael D.J. McNeill, Paul F. Lister, Codesign of Graphics Hardware Accelerators, SIGGRAPH/Eurographics Hardware Workshop, LA, 1997.

[10] Chabong Choi and Soonhoi Ha, "Software Synthesis for Dynamic Data Flow Graph", *8th IEEE International Workshop on Rapid System Prototyping*, June 24-26, North Carolina, 1997

[11] S. Ha, E. A. Lee, "Compile-Time Scheduling of Dynamic Constructs in Dataflow Program Graphs", IEEE Transaction on Computer, Vol. 46, No 7, July, 1997.

[12] Chanik Park, Jaewoong Chung and Soonhoi Ha, "Extended Synchronous Dataflow for Efficient DSP System Prototyping",10th IEEE International Workshop on Rapid System Prototyping(RSP), Florida, USA, June, 1999.

[13] S. Ha, E. A. Lee, "Quasi-Static Scheduling for Multiprocessor DSP", in *Proc. of ISCAS*, Singapore, June, 1991.

[14] Wonyong Sung and Soonhoi Ha, Efficient and Flexible Cosimulation Environment for DSP Applications, IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Special Issue on VLSI Design and CAD algorithms, Vol. E81-A, No.12, pp. 2605-2611, December.,1998.

[15] J.Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, "Ptolemy: a Framework for Simulating and Prototyping Heterogeneous Systems," *International Journal of Computer Simulation*, vol. 4, pp. 155-182, 1994

[16] http://www.arm.com/

[17] http://wwww.altera.com