Conversion of Reference C Code to Dataflow Model: H.264 Encoder Case Study

Hyeyoung Hwang Taewook Oh Hyunuk Jung* Soonhoi Ha
The School of Electrical Engineering And Computer Science
Seoul National University KOREA
* Samsung Electronics
{hyhwang, twoh, jung, sha}@iris.snu.ac.kr

Abstract – Model-based design is widely accepted in developing complex embedded system under intense time-to-market pressure. While it promises improved design productivity, the main bottleneck lies not in the design methodology but in constructing the initial algorithm representation in the specified model. It is particularly true if a complicated multimedia application is given in the form of a sequential reference C code. In this paper we propose a systematic procedure for converting a sequential C code to a dataflow specification that has been widely used in many design environments for DSP systems. The proposed technique is successfully applied to H.264 encoder algorithm as a case study.

I. Introduction

In HW/SW co-design methodology of embedded systems, system level specification enables us to model and analyze the system behavior in high level of abstraction to cope with the ever-increasing complexity of system design under relentless time-to-market pressure. Especially model-based specification is widely accepted because mapping the system behavior to processing components can be easily performed. In a model-based approach, the system algorithm is specified using a block diagram or a composition of function blocks. Once functional blocks are built and completely tested, they can be reused in many other systems thereby saving time and costs compared to traditional design approaches [1].

In this paper we use an extended synchronous dataflow (SDF[2]) model for algorithm specification. In SDF model, a node, or a block, represents a coarse grain function block whose body is described in C language. An arc represents a channel that carries streams of data samples from the source node to the destination node. The number of samples produced (or consumed) per block execution is called the output (or the input) sample rate of the block. In case the number of samples consumed or produced on each arc is statically determined and can be any integer, the graph is called a synchronous dataflow graph (SDF). A block is executable only after it receives the specified number of samples at all input ports. These restrictions make the model formal and data-driven. The SDF model has been widely used in many system-level design environments especially for digital signal processing systems [2][3].

The inherent difficulty of model-based approach lies in constructing the initial algorithm representation in the specified model, dataflow model in this paper. In most cases the algorithm description is given in the form of a reference C code. To get the benefits of model-based design approach, we have to convert the sequential reference code somehow to a dataflow specification. It is not a simple task for a system designer who may not know the algorithm details in particular. For instance, the reference C code of H.264 encoder algorithm [6] is about 32000 lines long and is scattered into 55 files.

In this paper, we propose a systematic procedure for converting a sequential C code to a dataflow specification. The procedure has been established after numerous hands-on experiences and successfully applied to H.264 encoder algorithm. It consists of three phases. First, we transform the reference code into the same code structure as would be automatically generated from a dataflow specification. In the second phase we identify the functional blocks from the transformed code and analyze their dependencies. At last we draw a dataflow graph, synthesize the sequential C code from the dataflow specification, and check the correctness and the performance of the code by comparing it with the reference code.

The rest of the paper is organized as follows. In Section 2, some background information is reviewed on the dataflow specification and the H.264 encoder algorithm. Section 3 presents the problem definition and the overview of the proposed solution. The detailed description of the procedure is explained in Section 4 with the H.264 encoder case study. Clustering and scheduling of the data flow specification are explained in section 5. The experimental results are presented in Section 6. Section 7 concludes the paper.

II. Background

A. Software Synthesis from Dataflow Model

Fig. 1 depicts the process of software code synthesis from an SDF specification [3]. A simple SDF graph in Fig. 1(a) contains three nodes, labeled A, B, and C. Each arc is annotated with the number of samples consumed or produced per node execution.

To generate a code from the given SDF graph, the order of block executions is determined at compile time, which is called "scheduling". Since a dataflow graph specifies only partial orders between blocks, there are usually more than one valid schedule. Fig. 1(b) shows a valid schedule. The parenthesized terms in schedule are used to express repetitive invocation patterns. These are called schedule

0-7803-9451-8/06/$20.00 ©2006 IEEE.

loops. In Fig. 1(b), the term 2(B(2C)) represents the invocation sequence BCCBCC. Each schedule loop is implemented as a loop structure in the synthesized code.

A code template according to the schedule is shown in Fig. 1(c). The function body of each function block is placed in the scheduled position. The code structure of the synthesized code from dataflow specification has the following characteristics.

- It may have nested loops.
- All function blocks appear in the main loop. A function block may not be called in another function block.
- State variables of function blocks may not be shared.

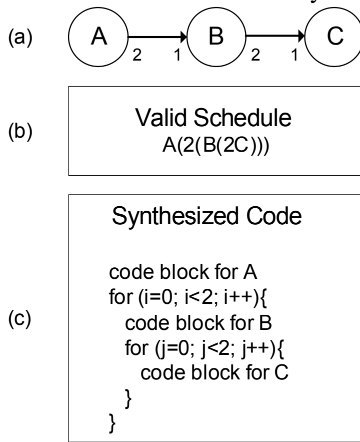

Fig. 1. An example of synthesized code from SDF

Fractional Rate Dataflow (FRDF) is an extension to the dataflow model in which fractional number of samples can be produced and consumed [4]. In the FRDF model, a constituent data type is considered as a fraction of the composite data type. For instance, a 16x16 macro-block is considered as 1/99 of a QCIF frame (176x144). Existing integer rate dataflow models can be easily extended to incorporate the fractional rates without loosing analytical properties. But it is reported that the FRDF model can reduce the buffer memory requirement in the synthesized code significantly, up to 70%, for some multimedia application.

B. H.264 Encoder Algorithm

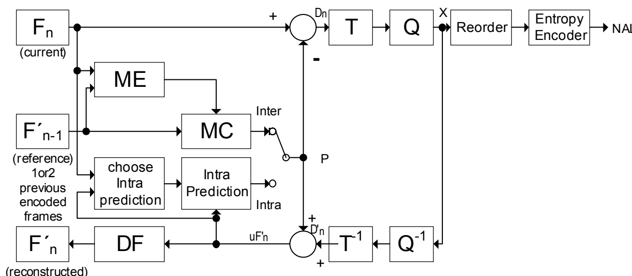

Fig. 2. H.264 encoder (baseline profile) block diagram

H.264 is a video coding standard (or Recommendation) made by the Video Coding Experts Group (VCEG) of the International Telecommunication Union Telecommunication Standardization Sector (ITU-T)[6]. While H.264 achieves bit rate saving ratio up to 50% compared to H.263+ and offers consistently good video quality at most bit rates, algorithm complexity grows significantly. More efficient compression and high quality video are attributed to the following features: Enhanced motion compensation, small blocks (4x4) for transform coding, improved in-loop deblocking filter, and enhanced entropy coding. The standard specification defines many options hence the reference code is very complex. In this work, we consider the baseline profile with a single slice mode for simplicity[7].

III. Problem Definition

Fig. 3 shows the skeleton of the H.264 reference code.

Fig. 3. Skeleton of H.264 reference code

As apparent from the Fig. 3 shown, the function call depth of the reference code is very deep and the key functions reside close to the bottom in the call graph. They should be elevated up to the top if they are to be defined as function blocks in the dataflow specification. While not shown in the Fig. 3, numerous global variables are defined and accessed

in various places. Therefore, it is not easy to grasp the data flow dependency between functions because data is coupled very tightly.

On the contrary, Fig. 4 illustrates the code structure of the synthesized code. So in the first phase of the proposed conversion procedure we transform the reference code to this code structure. The proposed code transformation consists of three steps that are applied repeatedly until no more transformation is needed. They are "Function restructuring", "Variable classification", and "Data sample rate decision" as shown in Fig. 5.

```
main(){
  ...
  ReadOneFrame()
  ...
  for(99){
    ...
    generate_mb()
    ...
    intrapred_luma_16x16()
    for(16){
      intra4_prediction()
      Intra4PredNSel()
      Intra4Dct4x4()
      Intra4Quant()
      Intra4cofACGen()          dct_luma()
      Intra4DeQ4x4()
      Intra4Idct4x4()
      Intra4PreReBlockGen()
      Intra4ReBlockGen()
    }
    inter()
  ...
}
```

Fig. 4. Transformed code skeleton

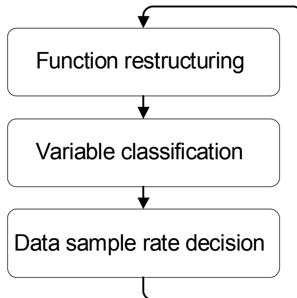

Fig. 5. Code transformation overview

In the Function-restructuring step, basic function blocks are identified and moved up to the top level in the call graph. In the Variable-classification step, the scope of variables are classified and analyzed to remove the global variables as much as possible. After this step, the redundant dependency between function blocks are removed so that the total ordering of function blocks is converted to partial ordering between function blocks. In the Data-sample-rate-decision step, we determine the sample rates of all function blocks. We use fractional rates as much as possible to reduce the memory requirements in the synthesized code. The process of transforming the reference code is repeated until the code structure of Fig. 4 is obtained.

IV. Code Transformation Techniques

In this section, we describe in detail the code transformation steps overviewed in the previous section. We also explain how the proposed technique is applied to the H.264 encoder example.

A. Function Restructuring

In this step we define 4 kinds of transformation. They are flattening, splitting, merging, and duplication, as demonstrated in Fig. 6.

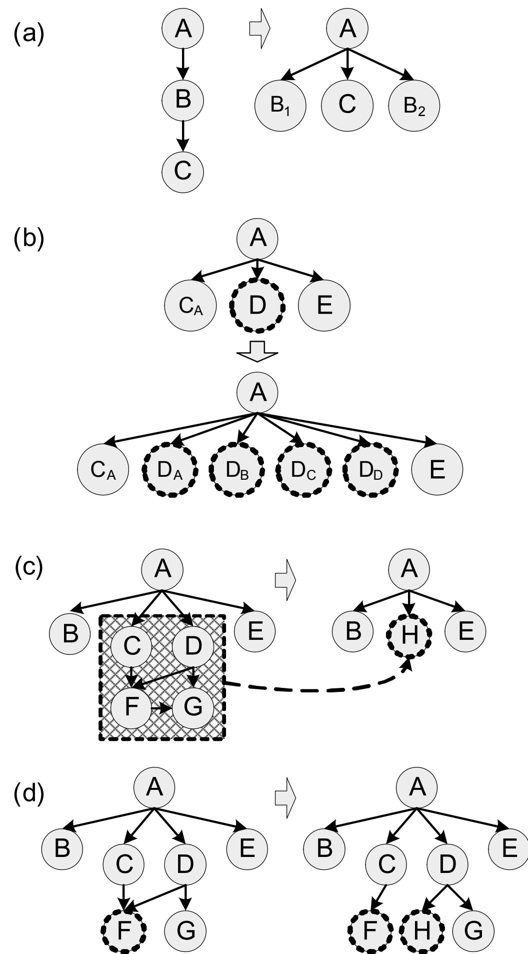

Fig. 6. Function restructuring transformations: (a) flattening, (b) splitting, (c) merging, (d) duplication.

Fig. 6(a) shows that flattening technique can be used to move a function up to one level higher. The caller function may need to be split into two parts that is called before and after the callee. In Fig. 3, the two functions *intrapred_luma()* and *dct_luma()* are called from inside *Mode_decision_for_4x4IntraBlocks()*. These two functions were first moved to same level as *intrapred_luma_16x16* as

shown in Fig. 4. Note that the function names of Fig. 4 are renamed to have a prefix, "intra4", for better readability.

Fig. 6(b) shows the case where a single function is split into multiple functions. The 5 different functionalities are included in function *dct_luma()*. The functionalities are dct, quantization, cofACGen, dequantization, and idct. So, as can be seen in Fig. 4, the *dct_luma()* is split and replaced with five functions.

Fig. 6(c) shows the opposite case where multiple functions are merged into a single function. When the call dependency is tightly coupled or data dependency between functions is tightly coupled, it is impossible to divide the functions into separate function blocks without re-programming. In this case, the tightly-coupled functions are merged into a single function to preserve the functionality of the code. In the H.264 encoder example, the functions for inter-prediction are tightly coupled so that we merge them into a single function block, called *inter()* in Fig. 4

Lastly, Fig. 6(d) shows the case where functions are duplicated. If the function F uses no shared variable or static variable inside, the function it can be duplicated without side-effect. Otherwise, two caller functions C and D should be merged by the merging transformation. When a function is duplicated, it should be renamed to avoid naming conflict. While function duplication has a drawback of increased code size, it increases the modularity and reusability of the function block.

B. Variable Classification

Variable classification is the most critical step to isolate the function blocks that are communicated with other function blocks only via port variables in dataflow specification. In the reference code, functions are tightly coupled with shared variables, so they are closely inter-dependent. The purpose of this step is to identify the true dependency between the functions by classifying the global and static variables.

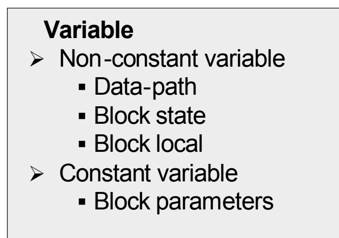

Fig. 7. Variable classification

As shown in Fig. 7, variables are classified into non-constant variables and constant variables. Non-constant variables, updated at run-time, create the dependency between functions. The non-constant variables are further classified into *data-path, block state*, and *block local* variables. They can be classified using the life time chart as illustrated in Fig. 8. The life time of a variable is defined as a set of durations what starts with a write operation and ends with the last read operation. Integer variable, *int a,* in Fig. 8 has multiple life durations since it is reused several times.

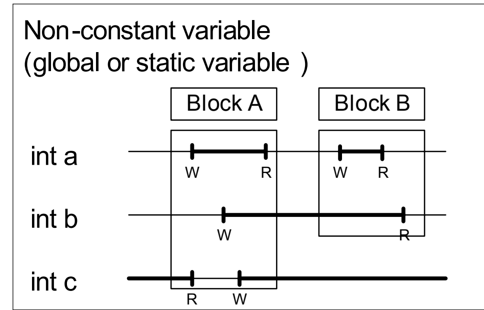

Fig. 8. Life time chart of non-constant variables

A data-path variable is a variable that is used as an interface between function blocks, especially at the top level, *i.e.* inside the main function. Integer variable *int b* in Fig. 8 is an example of a data-path variable. The life time of variable *b* starts at block A and ends at block B. A data path variable is translated to a port variable in the dataflow model.

Integer variable *int c* in Fig. 8 is not only read but also written during the execution of block A. The written value affects the next execution of block A. Therefore the variable *c* is classified as a block state variable and translated into a block state in the dataflow model.

Integer variable *int a* is a global variable that can be accessed from both block A and block B. But the value of *int a* inside one function does not affect the outcome of the other function. Hence, variable *a* can be classified as block local.

The constant variables are the variable whose value is not changed inside the main loop. They are translated into block parameters or global parameters without the risk of side-effect.

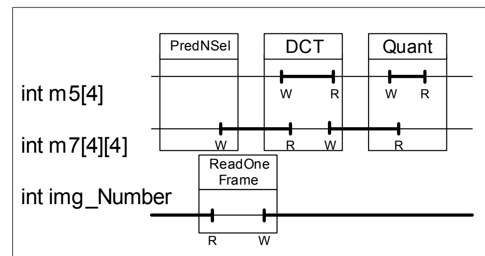

Fig. 9. Examples of variable classification

Fig. 9 shows the life time of some variables used in function *intra4x4()*. Variable *m5[4]*, which is shared between DCT and Quant functions, is classified to a block local variable. Variable *m7[4][4]* is a global variable used for sending the value written from the previous function to the next function – from PredNSel to DCT, and from DCT to Quant. Therefore, the variable is classified as a data-path variable. Lastly, variable *img_Number* is classified as a block state variable since the value modified at the previous

instance of the ReadOneFrame function affects the next instance of the function.

C. Data Sample Rate Decision

After function restructuring and data classification is completed, the sample rates of each function block are determined by examining how many data samples are produced and consumed at each port per function invocation. It also determines the execution frequency of function blocks. Fig. 10(a) shows a part of the translated code , which highlights the imgY_org data-port variable that connects two function blocks. By analyzing the data producing and consumption rates of two blocks, it is identified that one invocation of the ReadOneFrame block triggers the intra4_prediction block 16x99 times. Then we have three possible assignments of sample rates between two blocks. Note that a sample rate can be a fractional number.

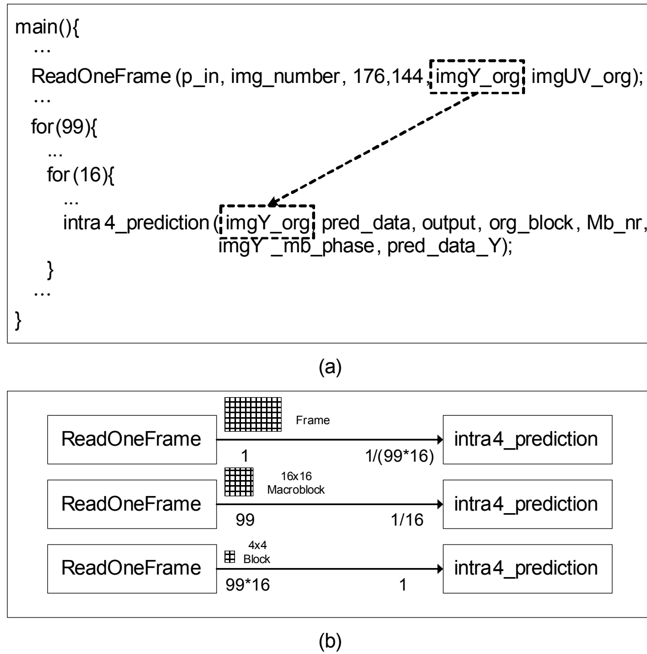

Fig. 10. Data sample rate decision between the ReadOneFrame and intra4_prediction blocks.

The first assignment in Fig. 10(b) indicates that the entire frame is passed from the *ReadOneFrame()* function to *intra4_prediction()*. In this case, the port buffer becomes *imgY_org[1]* which is a frame-type buffer whose size is 176x144 in the QCIF format. The data sample rate in this case is set to 1:1/(99x16). In the second assignment, the frame is broken down and passed in the unit of macroblock. In this case, the port buffer becomes *imgY_mb[99]* which is a macroblock array. In this case, the data sample rate becomes 99:1/16. Lastly, in the third example, the frame is passed in unit of 4x4 blocks. In this case, the buffer becomes *imgY_block[99*16]* and the data sample rate becomes 99*16:1.

Among three possible assignments we selected the second assignment to make a two-level nested loop as shown in Fig. 10(a). The corresponding dataflow subgraph of Intra4x4 prediction subsystem is shown in Fig. 11

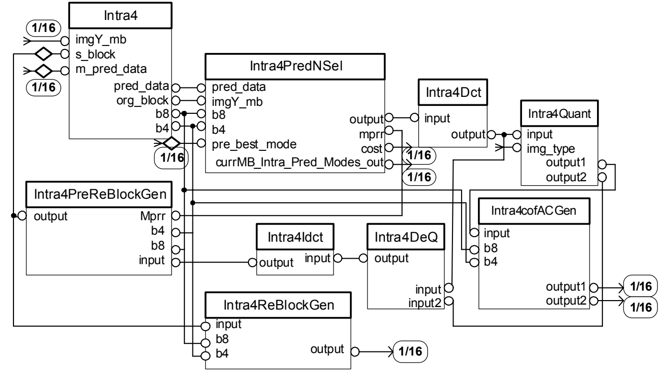

Fig. 11. Dataflow subgraph for Intra4x4 prediction subsystem

V. Clustering and Scheduling

After the conversion to a dataflow specification is completed, we have to find a valid schedule that generates the code structure as expected. Since obtaining an optimal schedule is beyond the scope of this paper, we outline how the valid schedule is constructed from the given dataflow specification. We make the clusters of function blocks that have the same sampling rates on the connected arc. Then the function blocks associated with intra4x4 prediction are merged together into a cluster. The Intra4x4 subsystem of Fig. 11 becomes a single cluster in the clustered graph. The cluster constructs a loop body in the final code as illustrated in Fig. 4.

The next step is the looping step of the clusters. If Intra4x4 subsystem is looped 16 times, the sample rates of this looped cluster become the same as the Intra16x16 subsystem. Then, we apply the merging step again to make it a hierarchically clustered graph. The second level loop of Fig. 4 is achieved in this way. We repeat the merging and looping steps until only one cluster remains at the top, which defines the main schedule body.

VI. Experimental Result

As a case study we convert the H.264 reference C code into an extended SDF model using our HW/SW codesign environment [8] that supports automatic C code generation from dataflow specification. Fig. 12 shows the schematic captured from the environment.

We first cut down the reference C code (JM original) to a light-weight version of the reference code, called Modified JM. The Modified JM code keeps the codes only for simple profile with single-slice mode. We then converts the Modified JM code to the dataflow graph applying the proposed methodology. From the dataflow specification, we obtain the synthesized code (Synthesized JM) from the design environment. We have compared these three codes in respect of code size, data size, and encoding time. All

experiments are done on a Linux platform (kernel version 2.6.8), and each C code is compiled using GNU gcc version 3.3.5.

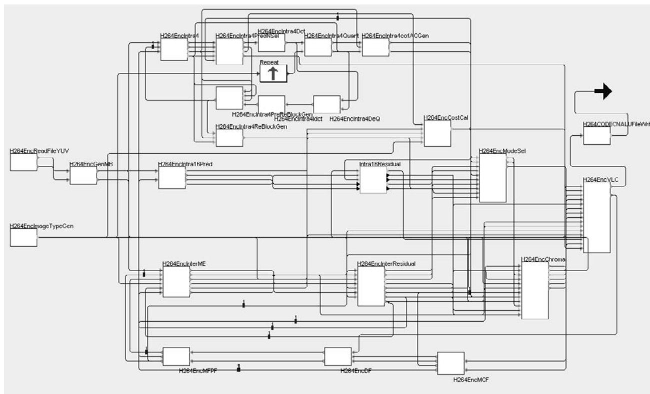

Fig. 12. SDF modeled H.264 encoder in PeaCE [8]

Table 1. Result on three different H.264 encoder codes

		JM original	Modified JM	Synthesized JM
Encoding time per Frame	I Frame	0.04 sec	0.04 sec	0.10 sec
	P Frame	0.70 sec	0.14 sec	0.31 sec
Code Size		369275 bytes	115863 bytes	148442 bytes
Data Size		7793536 bytes	2747900 bytes	2051908 bytes

Table 1 shows the experimental results on the three different H.264 encoder codes. As shown in the Table 1, the JM original code shows poor performance, and requires large code and data size, because it implements all options. So it is fair to compare the Modified JM code and the synthesized JM code. In this comparison, we do not perform any optimization on the dataflow specification.

As for the encoding performance, the Modified JM code shows about twice better performance than the synthesized code. This performance discrepancy is mainly due to data copying overhead between function blocks, which is the main target of optimization. In the current definition of function blocks, it includes several functions inside. Since these internal functions maintain the same variable sharing mechanism of the reference code, we explicitly copy the port buffers to the shared variables or vice versa. This copy overhead does not exist in the Modified JM code. It also causes code size expansion. It remains as a future work to remove this overhead since performance optimization is not a key objective of this paper.

On the other hand, the synthesized code shows better results than the Modified JM code in terms of the data size. This is because the Modified JM code allocates more frame –size variables than necessary. And the synthesized code allocates only a portion of frame for macro-block delivery between blocks and shares it through iteration, while the Modified JM code uses a frame-size variable. If we apply the buffer sharing optimization on the dataflow specification, the gap will be wider.

It took two man-months to transform the reference code to the code structure as shown in Fig. 4. It is performed by an expert in dataflow modeling, but he has only the basic knowledge on the encoding algorithm. So, two man-month includes the learning time of the H.264 encoder algorithm. It took another two man-month to draw a dataflow graph from the transformed code, which includes function block definition and code synthesis. It is performed by two graduate students who have no experience in this task.

VII. Conclusion

In this paper, we presented a systematic procedure to convert a sequential C code to a dataflow specification. The key technique is to transform the C code into a well-structured form for dataflow model. We experimented with an H.264 encoding algorithm to demonstrate how the proposed methodology can be applied to a complicated real-life multimedia application. The proposed technique includes function restructuring, variable classification, and data sample rate decision. These transformation techniques are applied in turn and repeatedly until the well-structured form is obtained.

We successfully obtain the dataflow specification and compared the synthesized code from dataflow specification with the reference code in terms of encoding time, code size, and data size. While the automatically synthesized code shows worse performance and code size, it shows better result on the data size. Optimization of the dataflow specification is left as a future work.

ACKNOWLEDGEMENTS

This work was supported by National Research Laboratory Program(number M1-0104-00-0015), Brain Korea 21, SystemIC 2010 Project and IT Leading R&D Project funded by Korean MIC. ICT and ISRC at Seoul National University provided research facilities for this study.

References

[1] K. Jerry, “Model-Based Design and Beyond: Solution for Today’s Embedded Systems Requirements,” *American Technology International*.

[2] E. A. Lee, D. G.. Messerschmitt, “Synchronous Data Flow”, *A Proceeding of the IEEE, Vol. 75, NO. 9, September 1987*.

[3] S. S. Bhattacharyya, P. K. Murthy, E. A. Lee, “Synthesis of Embedded Software from Synchronous Dataflow Specification,” *Journal of VLSI Signal Processing 21, 151-166(1999).*

[4] H. Oh, S. Ha, “Fractional Rate Dataflow Model for Efficient Code Synthesis.”

[5] S. Kwon, H. Jung, S. Ha, “H.264 Decoder Algorithm Specification and Simulation in Simulink and PeaCE.”

[6] H.264/AVC Software Coordination http://bs.hhi.de/~suehring/tml/

[7] Iain E. G.. Richardson, H.264 and MPEG-4 Video Compression, Willy, 2003

[8] PeaCE(Ptolemy extension as Codesign Environment) project homepage http://peace.snu.ac.kr/research/peace/