System Design for DSP Applications Using the MASIC Methodology

Abhijit K. Deb, Axel Jantsch, Johnny Öberg
Department of Microelectronics and Information Technology
Royal Institute of Technology, 164 40 Kista, Sweden
Email: { abhijit | axel | johnny } @ imit.kth.se

Abstract

Expensive top-down iterations are often required in the design cycle of complex DSP systems. In this paper, we introduce two levels of abstraction in the design flow by systematically categorizing the architectural decisions. As a result, the top-down iteration loop is broken. We also present a technique to capture and inject the architectural decisions such that the system models can be created and simulated efficiently. The concepts are illustrated by a realistic speech processing example, which is implemented using the AMBA on-chip architecture. Our methodology offers a smooth path from the functional modeling phase to the implementation level, facilitates the reuse of HW and SW components, and enjoys existing tool support at the backend.

1. Introduction

Process networks serve as the excellent basis for building ***functional models*** of DSP applications. They are, however, awkward for specifying the control logic needed to build complex systems, which often include SW components to achieve flexibility and HW components for performance critical parts. To address this problem, the Y-chart based DSP system design methodology employs a heterogeneous approach [1][2][3]. Other contemporary methodologies like the function architecture codesign [4], and the communication based design [5] tackle design challenges by separating functionality from architecture. The ideas in [5] are commercialized in the Cadence Virtual Component Co-design (VCC) tool. In general, all these methodologies employ the design flow of Figure 1(a).

In such a design flow, a system designer typically studies the functionality, takes the system specification, makes few initial calculations, and proposes an architecture to implement the functionality. Then the system performance is evaluated, for instance by simulation [1][2], and architectural decisions are altered to meet performance. However, the top-down iteration as shown in Figure 1(a) takes a lot of time, which hinders the design productivity.

To gain design productivity, our first objective is to break this top-down iteration into two smaller loops, and the second objective is to adopt a description formalism to capture and inject the architectural decisions efficiently.

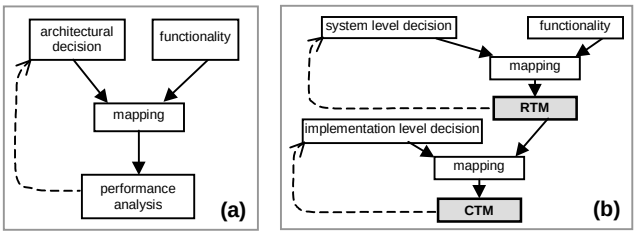

Figure 1: (a) Conventional system design flow (b) proposed design flow

In this paper, we present a systematic approach to classify the architectural decisions in two categories: *system level decisions* and *implementation level decisions*. The basis of this categorization is discussed in Section 2. We add system level decisions to a functional model to create an abstract intermediate model, called ***rate true model*** (RTM). Next, implementation level decisions are added to the RTM to build a ***cycle true model*** (CTM) of the transactions of the implementation architecture. Figure 1(b) shows that the formulation of the RTM between the functional model and the CTM breaks the top-down iteration and achieves the first objective.

The DSP system design methodology MASIC, short for *Maths to ASIC*, provides an elegant grammar based language to build abstract RTM-like system models [6]. The simulation and analysis technique used in MASIC is presented separately in [7]. To achieve our second objective, we have enhanced the MASIC language to be able to describe the architectural details of CTM.

The contribution of this paper is the improvement of the conventional design flow, which involves top-down iteration. Additionally, the grammar based language used in MASIC has been enhanced. Next, in Section 2 we present the background and basis of the systematic categorization of the architectural decisions. Our methodology is described in Section 3, followed by an illustrative example in Section 4. Section 5 presents the experimental results, and finally, we conclude this work in Section 6.

2. Background and Motivation

Functional modeling of signal processing applications usually begins using Kahn Process Network (KPN) [8] or different forms of dataflow networks like SDF [9], DDF [10], etc. Processes in a KPN are connected through infinite

1530-1591/04 $20.00 (c) 2004 IEEE

length point-to-point FIFOs. Graphically, processes are drawn as nodes and FIFOs as arcs. Processes read from the input FIFO when data is available (i.e., blocking read), perform computation in their private memory, manipulate their own state space, and write results in an output FIFO (non-blocking write). An important issue regarding an optimum implementation of these networks is to find a schedule to determine which process executes on which resource at which point in time. There exists an efficient technique of scheduling SDF networks for a sequential or parallel implementation [9]. Their approach also solves another practical problem, which is to find the sizes of the FIFOs, using a *balance equation* for each arc between the processes of a network [10]. Though, finding the FIFO size is not possible for the general case of KPN, there exists scheduling technique to find a reasonable upper bound using simulations with varying input data set [11].

The order and rate of process execution suggested by a schedule is not enough to build an RTM, We need two types of synchronization primitives. Firstly: scheduling assumes the ideal situation that processes have their local memories. Practical limitations, as would be discussed in Section 3.3, call for more synchronization before executing a process according to a schedule. Secondly: scheduling techniques, for example in [9], do not deal with the system interface to the environment and buffering of input data. Hence, the assumption of being able to schedule an input node at *any time* has to be synchronized with the availability of valid data at input. These synchronization primitives along with the schedule, and the finite FIFO size as required by the schedule is what we call the ***system level decisions***. This is the class of decisions where the designer has less freedom, as these decisions are dictated by the system interface, available resources, and a schedule for an optimum implementation of the processes on the resources. If performance requirements are not met then these decisions are changed. For example, the number and type of resources can be increased to add computational power. Again, as shown in [11], FIFO sizes can be increased to obtain a better schedule with higher performance.

The ***implementation level decisions*** are the class of decisions where the designer has the freedom to explore the design space. There is a wide design space related to the implementation of the communication architecture. For example, the FIFOs could be implemented using message passing or shared memory; the single address space of the shared memory could have centralized or distributed physical memory; the bus might have different widths, protocols and arbitration priorities, etc. Instead of buses a NoC based architecture, as proposed in [12], can be used to implement the communication architecture.

In our design flow, from functional model through RTM to CTM, the functionality remains the same. It is only the protocol of data transaction that evolves from abstract FIFO channels to bus protocols and component interfaces, like the bus functional models of embedded cores, etc. Therefore, to increase design productivity, we need an abstract way to describe the protocols.

There are different ways of describing a communication protocol. One way is to specify a state machine that implements the protocol using an HDL description. This is a low level approach as the designer has to describe the system using a finite state space and often needs to deal with the FSMs at the RT level.

A more abstract way is to specify the grammar of the protocol and synthesize a controller from it. There exist academic [13][14][15] and commercial [16] grammar based tools for protocol description. Though these approaches do not address the problem of system design, they demonstrate two clear advantages- firstly: the ease of protocol description using grammar, and secondly: the smooth path to HW synthesis from a grammar description. Inspired by these advantages, we have adopted the grammar based style in our methodology.

Abstract communication modeling has become a part of different system modeling languages like, SystemC [17] and SpecC [18]. However, we argue that grammars provide a more intuitive and natural way of describing protocols than the C++ or C language.

3. The MASIC Methodology

3.1 Functional Modeling

Modeling in MASIC begins at the functional level where individual DSP functions are developed in C or MATLAB like environment. These functions are connected by FIFO channels to constitute a network. At this stage, design issues are primarily algorithmic and verification is concerned with making sure that the specified signal processing figures of merits are met. The output of this level is a set of DSP functions in C without side effects.

3.2 The MASIC Description Language

Grammars are primarily used for pattern matching, and used in developing compilers for programming languages using tools like YACC [19]. We use grammars to recognize a particular signaling pattern, which represents a signaling protocol, and to produce the desired action when the pattern is seen at the input stream. The actions could be to generate a synchronization signal, store data in a buffer, or call a C function with the data saved in a buffer. There are two major sections of the MASIC description of a model:

- *Grammar rules*: used to describe protocols.
- *Constraints to the grammar rules*: used to specify architectural resources like FIFOs, synchronization signals, buses, memories, IOs (i.e., interface), etc.

The syntax of the MASIC grammar rule is shown in Figure 2. We have added an optional `clock_name` on top of the YACC-like grammar rule. Reading different streams at

```
(stream) [@clock_name] : [(condition)] pattern1 {action1}
                                     |  pattern2 {action2}
                                     | .. ..
                                     | reset {action-n}
                                     ;
```

Figure 2: General syntax of a grammar rule

different speeds symbolizes multiple clocks in a system and allows modeling of multi-rate systems. The rule in the figure says: a `stream` is read at the rate of the given `clock`. Next, it says, if a given `condition` is met and a certain `pattern` is seen at the stream, then the associated `action(s)` inside the curly braces would take place.

3.3 Rate True Modeling

We reuse the C functions developed at the functional level and add system level decisions using the MASIC description language to build an RTM. Constraints to the rules are used to declare the interface to the environment, FIFOs, synchronization signals, and the C functions. The invocation of each function is controlled by a grammar rule. The functions communicate over dedicated FIFO channels with atomic bulk transfer capability and timing is modeled only for major synchronization events. Several grammar rules read their input streams in parallel and have an inherent end recursion. Thus the model represents the concurrent non-terminating behavior of a system. The concurrent rules are arbitrated by an abstract controller, which provides scheduling and synchronization of events.

The abstract controller is built using MASIC description and it maintains the order and rate of function invocation as determined by a schedule. As mentioned in Section 2, more synchronization primitives are needed before a function can be invoked using a schedule. Let us consider the example SDF network shown in Figure 3(a). Here process `p1` reads one token from arc1, which has a delay of 2 unit; and produces one token in arc2, which has no delay. Corresponding values for the other processes and arcs are shown in the figure. Considering a run time of 1-unit for process `p1` and `p2`, and 3-units for process `p3`, an optimal schedule for the SDF network is shown in Figure 3(b). Here process `p1` and `p2` are running on processor1, and process `p3` is running parallelly on processor2. This schedule runs well if the arc1 has a FIFO depth of two.

Now, if processor2 does not have enough memory it would write directly to the output FIFO, (i.e., arc1). Since the size of arc1 is bounded, this operation would overwrite the FIFO and cause processor1 to read new data before it

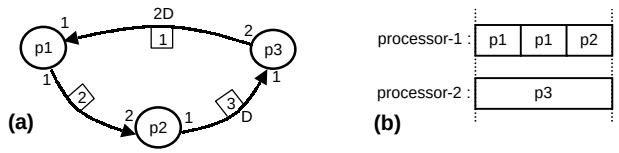

Figure 3: (a) An SDF network, (b) schedule for two parallel processors

has finished processing the old data. To synchronize them properly, processor2 has to wait until processor1 has finished reading tokens from arc1 (assuming processor1 has an input buffer), or until processor1 has finished executing process `p1` twice (assuming processor1 has no input buffer). We shall show how such synchronization can be easily expressed using grammar rules.

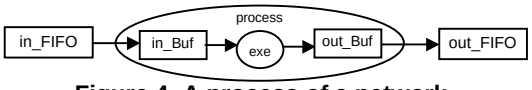

Figure 4: A process of a network

Let us consider the process shown in Figure 4 that reads data from `in_FIFO` and writes data to `out_FIFO`. It reads the `data_Rdy` and `room_Rdy` synchronization signals from the controller to know if there is data available in `in_FIFO` and if there is room available in `out_FIFO`, respectively. Assuming that the process has an input and an output data buffer, a grammar rule with two alternatives is shown in Figure 5(a). The first alternative specifies: if data is ready in `in_FIFO` then it would be copied to `in_Buf`; `read_Rdy` would be asserted so that another process can start writing to this FIFO; and a C function would execute on `in_Buf` and write result in `out_Buf`. The second alternative says: when there is room in the `out_FIFO`, the `out_Buf` is copied to `out_FIFO`; and the `write_Rdy` signal is asserted to indicate that there is new data available in `out_FIFO`.

Figure 5(b) is more interesting. In this case, we assume that the process shown in Figure 4 has neither input nor output data buffer. So, if it receives a `data_Rdy` signal it can not start executing the C function. However, if it receives both `data_Rdy` and `room_Rdy` signal, then it executes the C function directly on the data in `in_FIFO` and saves the result in `out_FIFO`.

```
(data_Rdy, room_Rdy) @clk:
      '1', '0'  { get(in_FIFO, in_Buf);
                  read_Rdy  <= '1';
                  call c_fun1(in_Buf, out_Buf); }
    | '0', '1'  { put (out_Buf, out_FIFO);
                  write_Rdy <= '1'; }
    | ...  ...
                                                    (a)
```

```
(data_Rdy, room_Rdy) @clk:
      '1', '0'  { null; }
    | '1', '1'  { call c_fun2(in_FIFO, out_FIFO);
                  read_Rdy  <= '1';
                  write_Rdy <= '1'; }
    | ...  ...
                                                    (b)
```

Figure 5: Example grammar rules

The addition of such simple rules keeps the order and rate of process execution as suggested by a schedule and adds the necessary synchronization to make the RTM work correctly with different implementation restrictions. The MASIC compiler reads the RTM and generates a VHDL description, which imports the DSP functions in C and performs a cosimulation using the Foreign Language Interface (FLI) of the VHDL. System simulation at this level only considers the computation delay. The computation delay of the C functions on a target architecture can be estimated easily when the run-time is data independent.

Even for the data dependent case, the computation time is bounded in hard real time applications. The estimated computation time is annotated in the RTM description using `wait` statement. The delay due to communication is unknown at this level.

3.4 Cycle True Modeling

In CTM, the dedicated communication channels of RTM are mapped to memories and shared buses, and the atomic bulk transfers between the synchronization points of the RTM are spread in time. The CTM does not alter the execution semantics of the RTM because the points of synchronization are maintained. The design step to create a CTM from an RTM involves the following tasks:

- Elaborating of the abstract communication channels into the detailed signaling mechanism required to express the bus protocol and the arbitration logic. If several functions are mapped on a single core, the channel between the functions needs to be implemented using the communication primitive offered by the operating system.
- Describing the interface of the hardware blocks onto which the functions are mapped to. To ease reuse of HW blocks we create the *Bus Functional Models* (BFM) of embedded cores and interface description of IP blocks. MASIC descriptions are used to build these models and they are saved in a library from where they are instantiated.
- Adapting the component interfaces to the bus protocol. We describe the glue logic between the pre-designed blocks and the bus architecture. The adapters can be saved in the library and reused. Currently the glue logic is written manually. However, this task can be automated, for example, using the approach presented in [20].

Communication architectures add significant amount of delay due to synchronization overhead [21]. Simulation of the CTM reveals these effects. If the performance is unsatisfactory then only the implementation level decisions need to be changed. This requires the design step from RTM to CTM to be performed. Thus the formulation of the RTM between functional model and CTM breaks the top-down iteration of the conventional design flow. The resulting design flow is shown in Figure 6. However, if changing the implementation level decisions are not enough to meet performance, then the initial mapping and scheduling needs to be changed. This is indicated by the dotted line in Figure 6. Next, using an example, we shall show how the implementation architecture can be described with ease using our grammar based technique.

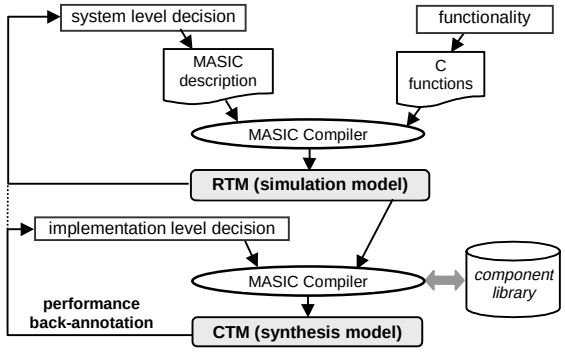

Figure 6: The complete design flow

4. An illustrative example

4.1 Functional model

Here we are using the Linear Predictive Coding (LPC) scheme. It samples input values at a rate of 8 kHz. The speech processing begins by buffering 160 input samples that corresponds to a 20 ms frame of input data. Then it performs the windowing operation on the samples and generates another 160 data. Next, the autocorrelation function takes this result and computes 11 autocorrelation lags, which the LPC block reads to compute 10 coefficients. Finally, the reflection coefficients are computed from the LPC values. The outputs of this phase are four C functions that compute the windowing (`win`), autocorrelation (`cor`), LPC (`lpc`) and reflection (`rfl`) coefficients.

4.2 Rate True Model

We decide to implement our four DSP functions on four MIPS32-4K processor cores. For the application at hand, we use SDF network scheduling technique, which also gives us the FIFO depth. Since the chosen processor has enough data cache, we do not need the kind of synchronization shown in Figure 5. However, few synchronization primitives are needed to handle the input data buffering and startup sequence. The interface specification of the system requires sharing the `in_port` for downloading the windowing coefficients at startup, and then starting regular processing of data. Hence the startup sequence would look like: reset, downloading coefficients, and then regular data processing.

We add these system level decisions and reuse the C functions to build an RTM of the system, as shown in Figure 7. The C functions, interface to the environment, and the FIFO channels among the functions are declared using grammar constraints. The grammar rules are used to describe the order of process execution along with necessary synchronization. MASIC description of abstract RTM-like model has been shown elaborately in [6]. Though this model captures the major synchronization events, it is still a fairly abstract model, as the implementation dependent complex protocols of data transactions have not been included yet. As a result the simulation of RTM is much faster than that of the CTM.

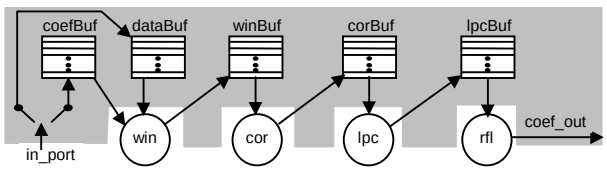

Figure 7: Rate true model of the LPC codec

4.3 Cycle True Model

We decide to realize the RTM using the AMBA on-chip architecture. The functional blocks would become the bus masters and the buffer instances would be mapped to the slave units. The abstract channels of the RTM are elaborated according to the AMBA AHB specification [22], and the interfaces on the DSP blocks are elaborated to the BFM of the MIPS32-4K core [23]. A simplified view of the architecture is shown below. The data and control lines are shown in solid and thin arrows, respectively. The request and grant lines between masters and arbiter are not drawn.

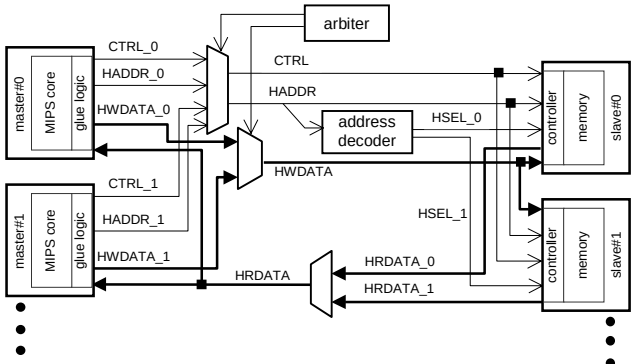

Figure 8: Simplified view of the AMBA architecture

The AMBA AHB uses separate read and write buses operating through a centrally multiplexed architecture. To gain performance, it works in a pipelined fashion where the address-phase of a transfer proceeds simultaneously with the data-phase of the previous transfer. In this section we shall show how these complex protocols can be described using our grammar based technique.

The control, address and data buses are represented as internal signals using grammar constraints. We connect the read and write data buses to the read and write data ports of the core BFM. The address and control information of the master, which wins the arbitration, are propagated to the slaves through the address bus HADDR. The address decoder, shown in Figure 8, selects a slave by combinational decode of the higher bits of the address bus HADDR. Figure 9 shows the grammar rule for the address decoder. The clock information is absent in the grammar rule, which symbolizes a combinational behavior, and the decoding logic is described using the pattern-action pairs.

Depending on the slave select signal, the appropriate slave unit is selected and the control signal CTRL tells the type of transaction that needs to be performed. The aggregate signal CTRL is composed of several AHB signals like HWRITE, HTRANS, HBURST, etc. Figure 10 shows the grammar description involved in the slave module that includes a memory and a controller. The memory behavior is described as: if the condition `CS` is true, then a `'1'` at `RWS` causes a write and a `'0'` at `RWS` causes a read. The next line fetches the address from the address-phase of the transfer. It says to read the address (HADDR) at the arrival of the HCLK if the HSEL signal is high, which is the address-phase. By default, the clock is implemented as rising edge triggered and the reset as an asynchronous reset.

Next, the slave controller reads the aggregate control word, separated by commas. If the first bit pattern is seen, then it asserts the chip select signal and de-asserts the RWS so that the RAM outputs (i.e. a read operation) a single word. The second pattern causes a write operation. The third pattern initiates a burst read of unspecified length. The first transfer of a burst uses `HTRANS="10"`, followed by `"11"` for the remaining transfer and terminates with a `"00"`. This whole information is described as follows: if a pattern of `('1', '0', "10", "001")` is seen, the first data of the burst is supplied and then it looks for a pattern labeled as `branch1`. The `branch1`, as described below, is repeated until there is a `"11"` at the `HTRANS` input; the other inputs are don't cares, represented by the `'-'`s. Finally, `branch1` terminates when a `"00"` is seen at `HTRANS`.

Similarly, the rest of the architecture and the glue logic for the MIPS cores are described. We avoid the state based description and explicitly describe the transactions, from where the compiler generates the controller in VHDL.

```
-- grammar rule for the address decoder
(HADDR_HIGH_BIT) : "00"  { HSELv <= "0001"; }
                 | "01"  { HSELv <= "0010"; }
                 | "10"  { HSELv <= "0100"; }
                 | "11"  { HSELv <= "1000"; }
                 ;
```

Figure 9: MASIC description of address decoder

```
-- the memory block of slave_0
(RWS) : (CS)  '1' { MEM(ADDR) <= HWDATA; }
              | '0' { HRDATA_0  <= MEM(ADDR); }
              ;


-- fetching the HADDR at the rising edge of the HCLK
-- of the address cycle
(HSEL) @HCLK : '1' { ADDR <= HADDR };


-- Protocol of AMBA Slave Controller
(HSEL, HWRITE, HTRANS, HBURST) @HCLK
      : '1', '0', "10", "000"  { CS     <= '1';
                                 RWS    <= '0';
                                 HREADY <= '1'; }
      | '1', '1', "10", "000"  { CS     <= '1';
                                 RWS    <= '1';
                                 HREADY <= '1'; }
      | '1', '0', "10", "001"  { CS     <= '1';
                                 RWS    <= '0';
                                 HREADY <= '1'; }
         branch1

      | '1', '1', "10", "001"  { CS     <= '1';
                                 RWS    <= '1';
                                 HREADY <= '1'; }
         branch2

      | reset                  { CS     <= '0';
                                 RWS    <= '0';
                                 HREADY <= '0'; }
      ;

branch1  : '-', '-', "11", '-' { CS     <= '1';
                                 RWS    <= '0';
                                 HREADY <= '1'; }
            branch1

         | '-', '-', "00", '-'
         ;

branch2  : '-', '-', "11", '-' { CS     <= '1';
                                 RWS    <= '1';
                                 HREADY <= '1'; }
           branch2

         | '-', '-', "00", '-'
         ;
```

Figure 10: MASIC description of the slave

5. Results

The experiments are performed using two examples: the LPC codec described in the previous section and a ΣΔ demodulator. The ΣΔ demodulator has two FIR filters of length 31 and 69, one integrator and one differentiator. The DSP functions are developed in C during the functional modeling phase, and reused to build an RTM, and then an AMBA based CTM. The BFMs of MIPS32-4K cores are used as masters. Table 1 compares the code size of the MASIC description of the RTM and CTM, and the VHDL description of the CTM. The increase in productivity in term of the design-hour could be guessed from the bulk of VHDL code and the number of states. In the MASIC approach, the system transactions are expressed in abstract grammar, from where the VHDL is generated by the MASIC compiler.

Table 2 shows the simulation time of these designs for 1 sec of input data. The RTM simulates much faster than the CTM, as the RTM does not use any intricate signaling protocol. Such speedups are highly beneficial considering the iterative nature of the design flow.

The VHDL description of the CTM of the LPC design is synthesized using the Synopsys Design Compiler. The cell area was 3784 gates using the *lsi_10k* as target technology. This area does not include the cores or the memory cells. Those were used as black boxes during synthesis.

Table-1: Code size and number of states

	Word count	
	LPC	ΣΔ
MASIC description of RTM	305	276
MASIC description of CTM	2998	2437
Cycle true VHDL description	15865	11952
Number of states in VHDL model	176	142

Table-2: Simulation time

Generated VHDL	Simulation time	
	LPC	ΣΔ
RTM	13 min 47.7 sec	8 min 36.1 sec
CTM	51 min 41.2 sec	37 min 23.5 sec

6. Conclusion

We have presented a methodology that breaks the top-down iteration of the conventional design flow. It takes less time to create and simulate the RTM, which is highly advantageous in an iterative design flow. Though the work is described in the context of the MASIC methodology, any DSP system design methodology would benefit from the systematic formulation of the intermediate abstraction level of the RTM. In addition, we have enhanced the MASIC language to be able to describe the details of the CTM.

So far only SDF examples have been considered. However, the methodology is not restricted to SDFs. To model a process network in general a dynamic scheduler needs to be built. Since the RTM is meant for efficient simulation, we are considering the compilation of the MASIC description of RTM to SystemC, instead of VHDL.

7. References

[1] B. Kienhuis, Ed Deprettere, K. Vissers and P. van der Wolf, "An Approach for Quantitative Analysis of Application Specific Dataflow Architectures," in *Proc. IEEE Conf. Application Specific Systems, Architectures and Processors*, pp. 338-349, Jul. 1997.

[2] P. van der Wolf, P. Lieverse, M. Goel, D.L. Hei and K. Vissers, "An MPEG-2 Decoder Case Styde as a Driver for a System Level Design Methodology," in *Proc. CODES*, pp. 33-37, May 1999.

[3] E.A. de Kock et al., "YAPI: Application Modeling for Signal Processing Application," in *Proc. DAC.*, pp. 402-405, Jun. 2000.

[4] M. Sgroi, L. Lavagno and A. Sangiovanni-Vincentelli, "Formal Models for Embedded Systems Design," *IEEE Design & Test of Comp*., vol. 17, no. 2, pp. 14-27, Jun. 2000.

[5] K. Keutzer, S. Malik, A.R. Newton, J.M. Rabaey and A. Sangiovanni-Vincentelli, "System-Level Design: Orthogonalization of Concerns and Platform based Design," *IEEE TCAD*, vol. 19, pp. 1523-1543, Dec. 2000.

[6] A. Hemani, Abhijit K. Deb, J. Öberg, A. Postula, D. Lindqvist and B. Fjellborg, "System Level Virtual Prototyping of DSP SOCs Using Grammar Based Approach," *Kluwer Design Automation for Embedded Systems*, vol. 5, no. 3, pp. 295-311, Aug. 2000.

[7] A. Hemani, A. Postula, Abhijit K. Deb, D. Lindqvist and B. Fjellborg, "A Divide and Conquer Approach to System Level Verification of DSP ASICs," in *Proc. IEEE Int. High Level Design Validation and Test*, pp. 87-92, San Diego, California, Nov. 1999.

[8] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Proc. IFIP Congress '74*, pp. 471-474, Aug. 1974.

[9] E.A Lee and D.G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Trans. Comp.*, vol.C-36, no. 1, pp. 24-35, Jan. 1987.

[10] J.T. Buck and E.A. Lee, "Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model," in *Proc. Int. Conf. Acoustics Speech & Signal Processing*, pp. 429-432, vol. 1, Apr. 1993.

[11] T. Basten and J. Hoogerbrugge, "Efficient Execution of Process Networks," in *Proc. Communicating Process Architectures*, pp. 1-14, IOS Press, Amsterdam, 2001.

[12] S. Kumar et al., "A Network on Chip Architecture and Design Methodology," in *Proc. IEEE Comp. Society Annual Symposium on VLSI*, pp. 105-112, Apr. 2002.

[13] A. Seawright, F. Brewer, "Clairvoyant: A Synthesis System for Production-based Specification," *IEEE TVLSI*, vol. 2 no. 2, pp. 172-185, June 1994.

[14] J. Öberg, A. Kumar, and A. Hemani, "Grammar-Based hardware synthesis from port size independent specifications," *IEEE TVLSI*, vol. 8, no. 2, pp. 184-194, April 2000.

[15] R. Siegmund and D. Müller, "Automatic Synthesis of Communication Controller Hardware from Protocol Specifications," *IEEE Design & Test of Comp.*, vol. 19 no. 4, pp. 84-95, Jul-Aug. 2002.

[16] A. Seawright, U. Holtmann, W. Meyer, B. Pangrle, R. Verbrugghe, J. Buck, "A system for compiling and debugging structured data processing controllers," in *Proc. Euro DAC.*, pp. 86-91, Sept. 1996.

[17] T. Grötker et al., *System Design with SystemC*, Kluwer Academic Publishers, Norwell, MA, 2002.

[18] R. Dömer, D.D. Gajski and A. Gerstlauer, "SpecC Methodology for High-Level Modeling," in *Proc. 9th IEEE/DATC Electronic Design Processes Workshop*, Monterey, CA, Apr. 2002.

[19] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, MA, 1986.

[20] R. Passerone, J.A. Rowson and A. Sangiovanni-Vincentelli, "Automatic Synthesis of Interfaces between Incompatible Protocols," in *Proc. DAC.*, pp. 8-13, Jun. 1998.

[21] K. Lahiri, A. Raghunathan and S. Dey, "System-Level Performance Analysis for Designing On-Chip Communication Architectures," *IEEE TCAD*, vol. 20, no. 6, pp. 768-783, Jun. 2001.

[22] AMBA on-chip bus specification [Online], http://www.arm.com

[23] MIPS32 4K Processor Core Family Integrator's Manual, [Online], http://www.mips.com