C-Based SoC Design Flow and EDA Tools: An ASIC and System Vendor Perspective

Kazutoshi Wakabayashi, *Member, IEEE*, and Takumi Okamoto

***Abstract*—This paper examines the achievements and future of system-on-a-chip (SoC) design methodology and design flow from the viewpoints of an in-house electronic design automation team of an application-specific integrated circuit and system vendor. We initially discuss the problems of the design productivity gap caused by the SoCs complexity and the timing closure caused by deep-submicrometer technology. To solve these two problems, we propose a C-based SoC design environment that features integrated high-level synthesis (HLS) and verification tools. A HLS system is introduced using various successful industrial design examples, and its advantages and drawbacks are discussed. We then look at the future directions of this system. The high-level verification environment consists of a mixed-level hardware/software co-simulator, formal and semi-formal verifiers, and test-bench generators. The verification tools are tightly integrated with the HLS system and take advantage of information from the synthesis system. Then, we discusses the possibility of incorporating physical design feature into the C-based SoC design environment. Finally, we describe our global vision for an SoC architecture and SoC design methodology.**

***Index Terms*—C-based design, dynamic reconfigurable computation, floor plan, hardware description language, hardware/software co-simulation, high-level synthesis, high-level verification, SoC, timing closure.**

I. Introduction

THERE are two major problems that designers encounter with the rapid progress of VLSI process technology. The first is the productivity gap between VLSI complexity and design productivity, of which increasing rates are said to be 58% and 21% per year, respectively [1]. Moreover, even though chips are growing in size, the time it takes to design for specific circuit sizes is being reduced by about 30% annually because of the recent short life cycle of chips. Therefore, the average duration of 52 weeks (i.e., 24 weeks for system and logic design, eight weeks for back-end tools, four weeks for prototyping, and 16 weeks for evaluation) for a 2M gate circuit in 1999 needs be reduced to about 30 weeks in 2000 and 12 weeks in 2001. (These absolute figures are just standards for various cases, since design productivity varies by a factor of ten according to the designs and the designers.) In reality, however, more than 80% of chips cannot meet the original development schedule even now, since the schedule tends to be decided based on a time-to-market plan and not by summing up the necessary time required for each design process. To meet such a short design deadline, more designers (man-power) are being committed to projects, although productivity is not proportional to the number of designers. The design cost for a specific number of gates (e.g., 100 gates) is also increasing at a rate of 30% per year according to the growth of the chip size. We aim to decrease the power consumption of the chip at a rate of 30% per year, even though the chip size is increasing.

Even though the duration of projects, the number of man-hours, chip costs and the power consumed have all increased as chip size has increased, we have to find ways to reduce all of these. The past VLSI design crisis, or the production gap between the producible gate number and the number of gates that can be designed, has already been solved by new-generation EDA tools. In the late 1970s through the early 1980s, automatic placement and routing tools dramatically improved the layout process. Then, from the late 1980s through the 1990s, logic synthesis tools improved both the design duration and the design efficiency shown in Fig. 1. These automations have enabled designers to handle fewer components at the implementation level in each era. Without such EDA innovation, design productivity would not be able to keep up with the increasing complexity of VLSIs, since design periods needs to be shrunken in a faster pace than the chip size grows. We believe that many designers will have to raise their implementation level to the behavior level and should use a high-level synthesis (HLS) system [2] (or a behavior synthesis system) at the beginning of the twenty-first century.

In addition, the severe time-to-market pressure prevents design reuse. Recently, designers prioritize time-to-market first, design optimization second, and design cost third. Design for reuse has lower priority, since designing reusable modules is more difficult and takes longer. Nevertheless, designers have to finish their on-going design as soon as possible. Therefore, we encourage designers to use existing modules, but more than 60% of modules are designed from scratch at our company. Physical intellectual property (IP) reuse is not common at NEC—with the recent exception of CPU cores or memories. However, synthesizable register transfer level (RTL) IPs have become popular, but there are many problems with these, such as flexibility of functions, performance, and timing. We believe that behavioral-level IPs with an HLS system are more promising than RTL IPs.

The second problem comes from deep-submicrometer (DSM) technology. In DSM, the interconnect delay has become larger than the gate delay, so the delay estimation at the gate level is not accurate at all. Even after layout, accurate estimation for the interconnect delay has become difficult

Manuscript received May 22, 2000. This paper was recommended by Associate Editor R. Camposano.

K. Wakabayashi and T. Okamoto are with C&C Media Research Laboratories, NEC, Kawasaki, Japan (e-mail: wakaba@ccm.cl.nec.co.jp; okamoto@ccm.cl.nec.co.jp).

Publisher Item Identifier S 0278-0070(00)10453-1.

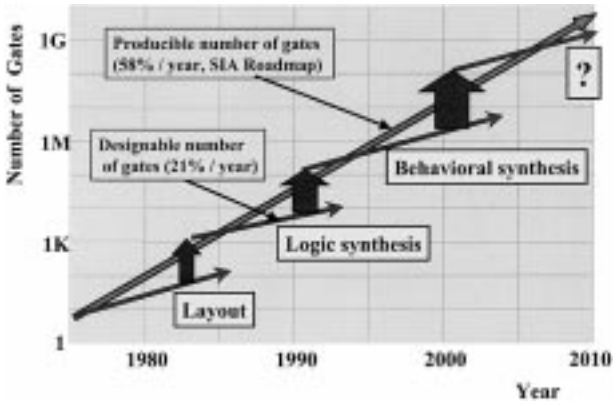

Fig. 1. Design crisis and CAD innovations.

because we cannot use the simple delay calculation formula we used before; we now need to take into account the resistance, inductance, and capacitance to calculate the delay. When we perform behavioral and logical synthesis and automatic layout in sequence, the timing closure after layout becomes a major problem. In addition, signal integrity (SI) problems such as cross-talk and IR voltage drop, and reliability problems such as electro migration and hot electrons become serious problems. These DSM problems cause more chip rework after the first tape-out.

Since we began to use various kinds of design verification tools, the first silicon chip has exhibited lower numbers of logical bugs and silicon re-spins, however, DSM problems have recently created the need for more silicon spins.

Although DSM problems have to be solved mainly from the semiconductor technology side, the design technology side has to find some appropriate design tactics to avoid these DSM problems. As discussed above, we have to cope with increasing complexity of system-on-a-chip (SoC) and the timing closure problem for DSM. We will discuss how we have handled these problems to date and then present our vision for the near future. The rest of this paper is organized as follows: Section II proposes a C-based SoC design flow with integrated EDA tools. Section III presents our HLS systems with various synthesis examples, and discusses their merits, problems and future directions. Section IV describes our high-level verification environment consisting of a co-simulator, formal and semi-formal verifiers, and test-bench generators. Section V gives our view on the SoC design language. Section VI describes the timing closure problem in physical design. Section VII gives our perspectives on next-generation SoC design and Section VIII concludes the paper.

II. Target SoC Design Flow with Integrated EDA Tools

Our integrated SoC design environment shown in Fig. 2 consists of hardware synthesis flow, software design flow, and verification flow for the entire SoC system and hardware modules. The main feature of our configuration is that the tool flow is based on a high-level synthesis (HLS) system, called "**Cyber**" [7], [9], [8]. Other tools take advantage of information from the HLS system and the characteristics of the target architecture of the synthesis system. In our flow, the designer describes the C description not only as a "specification or modeling language," but also as a final "implementation language," since we would like to avoid double coding for simulation and synthesis.

Initially, a designer verifies a pure algorithm for SOC in C language. Then, he/she divides it into software and hardware descriptions in C and behavior description language (BDL), which is our original extended C language for hardware behavior descriptions. In hardware design flow, designers use a behavioral IPs, as will be explained later. The Cyber-HLS system transforms the BDL description into RTL-hardware description language (HDL) descriptions such as verilog and VHDL. The equivalence between the BDL and RTL descriptions are guaranteed by our theorem prover-based verifier, and some given properties for each module are verified by our model checker. The generated RTL is passed to a logic synthesis and automatic placement and routing tools. We are trying to connect the HLS system and our Floorplanner in order to achieve a good performance circuit in DSM chips.

The entire system containing hardware and software parts is simulated in our system level co-simulator called "**ClassMate**" [10]. It can simulate the behavior and cycle-level model. With this co-simulator, hardware and software can be designed simultaneously. In our C-based design environment, all simulation models for hardware and software are described in C(C++) language. Cycle-level hardware simulation models are generated by the Cyber HLS system.

In software design flow, an embedded application program is described in C, and the description is compiled into assembly codes, which are executed in a CPU model described in C(C++). Our C-based system co-simulator runs several-hundred times faster than a commercial RTL-HDL-based co-simulator, so it can be applied to a SOC with several million gates.

III. High-Level Synthesis

This section discusses various aspects of HLS in terms of this industrial design experience and suggests future directions for HLS system and design methodology.

A. Control-Dominated Circuits

As RT-based design is based on a structural design methodology, it is not a good idea to design a complex sequential controller at the RT level. Before the logic synthesis era, designers created a sequencer by modifying various types of counters, and since the introduction of logic synthesis they have tried to avoid a large finite state machine (FSM) controller in wire logic because it causes logical bugs and a long delay. If control is too complicated, they like to utilize a software or a firmware controller. Currently, HLS enables designers to apply wired logic to such complicated control-dominated circuits. In fact, until now our HLS system has been applied to control-dominated circuits more than data intensive circuits. To control dominated circuits, manual scheduling mode, which enables the designer to put a clock boundary symbol (or statement) at appropriate places of their/her behavioral description, is used more often than the automatic scheduling mode. This is because control-dominated circuits communicate to other circuits at almost every cycle and,

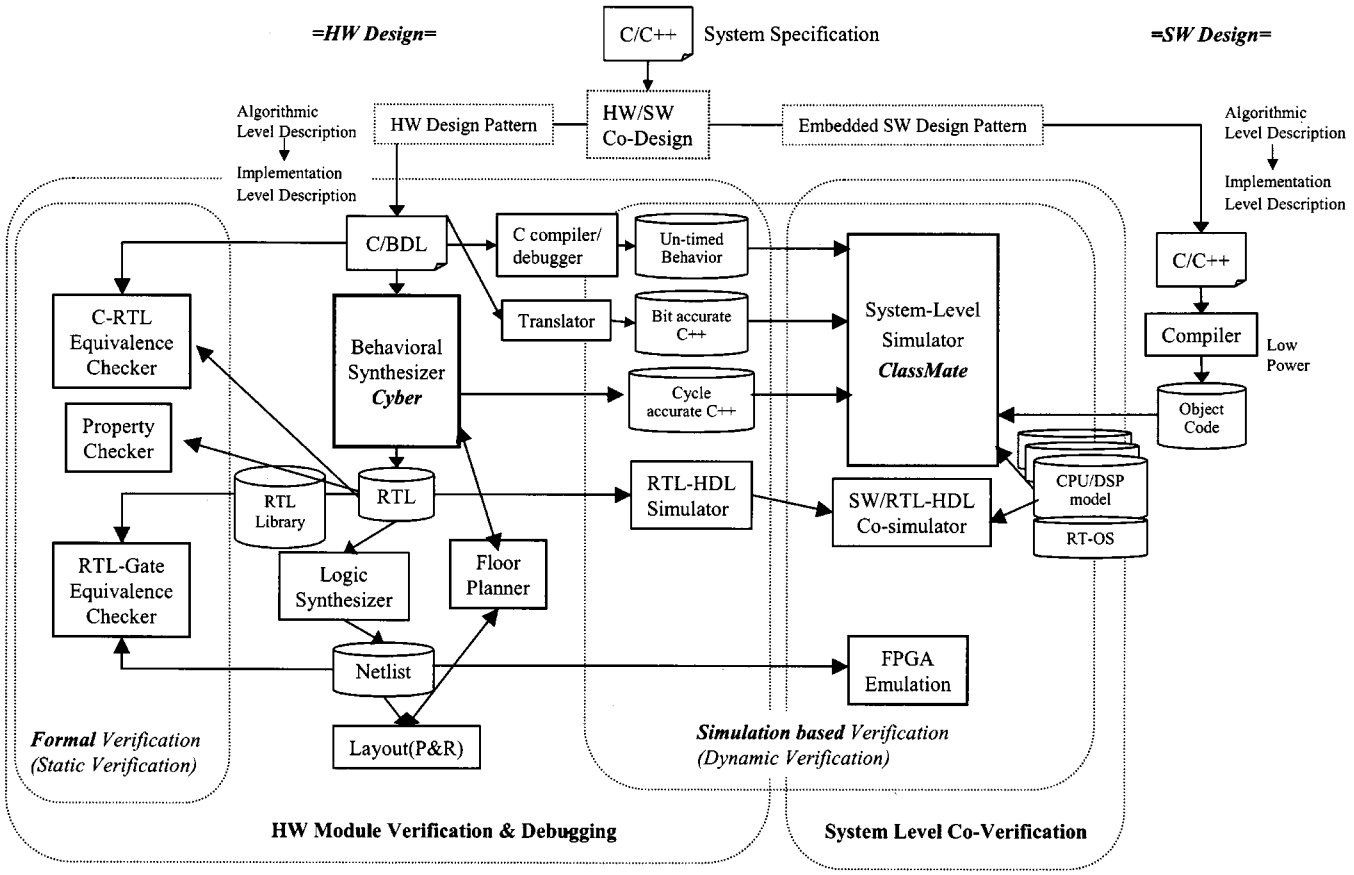

Fig. 2. C-based SOC design environment.

therefore, there is not much freedom for automatic scheduling. With the manual scheduling mode, it is easier to modify the control sequences according to the specification changes.

1) Firmware vs. Wired Logic: A typical example of a control-dominated circuit is a Network Interface Card (NIC) chip. We designed a NIC chip for PC clusters based on the "Virtual Interface Architecture" that has a 1.25-Gb/s communication speed with our HLS system Cyber in 1998. The behavioral description is about 20K lines. The resulting circuit contains more than 20 parallel communicating processes and more than 1000 states. The largest FSM has about 300 states and the total size is 150K gates.

Designing a 1,000-state controller at the RT level is not a good idea since this level is too complex for fixing bugs or modifying the control. Therefore, designers have tried to use firmware (microcode) control with some custom-made processors, but according to the preevaluations the speed of 1.25 Gb/s is a little too fast. Consequently, designers applied our HLS system to synthesize a wired logic controller. The results have been fairly encouraging. The gate size is 40% smaller and the speed is 30% faster than what were expected. They also found that behavioral design is much easier than firmware (microcode) design since they have to pay attention to the RTL structure; i.e., number of registers, busses and connections among them. They admitted that modifying the cycle behavior at the behavioral level is actually an easy task, although it is very difficult at the RT level.

This example shows that embedded softwares on embedded processors can be replaced without much difficulty with wired logics by using our HLS tool. The design cost of both is not that different and the wired logic controller surpasses a firmware controller in terms of performance and power consumption. Programmability should be achieved by use of reconfigurable chips such as field programmable gate arrays (FPGAs).

2) Partitioning of a Large FSM into Parallel FSMs: The wired logic for a large FSM usually has large delay for state decoding. Many state encoding techniques have been proposed and they are implemented in logic synthesis systems, but their delay optimization takes long time and does not work well enough for a large FSM in the sense of controllability for area/speed tradeoff. Our HLS system partitions such a large FSM into an arbitrary number of same-sized FSMs, and this enables designers to select the best partitioning according to delay and area. The partitioning of a large FSM is also good for low-power design as well as for low delay. The partitioning of an FSM with large and complicated control is almost impossible for designers. In this sense, the HLS system is superior to manual RTL design when designing a large sequential circuit.

3) Optimizing Gate Level Delay at the Behavioral Level: Many designers are worried about design closure, especially timing closure when they use HLS, since the behavior level is too distant from the gate level (or layout level) to control circuit characteristics. For example, they are afraid it becomes more difficult to optimize gate level delay with changes of behavior and/or HLS options than changes of RTL description and/or logic synthesis options. However, we have found that these concerns are not always valid. Eliminating

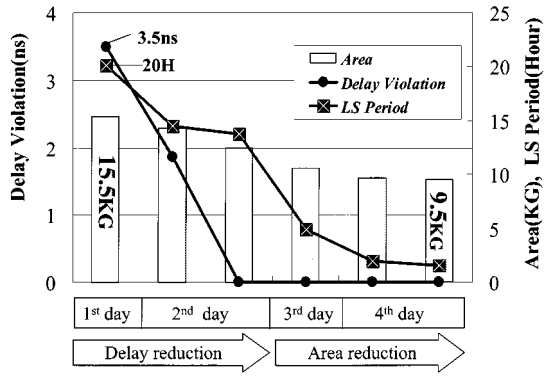

Fig. 3. Delay optimization process.

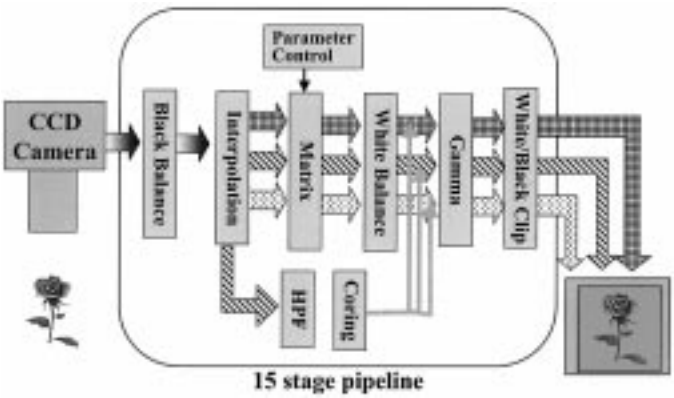

Fig. 4. Configuration of CCD video processor.

delay violation is one of the most difficult problems for logic synthesis tools even if we do not care the interconnect delay after layout. The HLS system helps designers to cope with this problem. Fig. 3 shows the delay and area optimization process when modifying behavior (i.e., clocking boundaries) and/or HLS options (i.e., behavioral re-timing and FSM partitioning). On the first day, the circuit violated its delay constraint by 3.5 ns after a 20-h run of logic synthesis. Logic synthesis introduced buffers, powered gates, and duplications of logic gates to reduce the delay, but none of these worked. With our HLS tool, the designer generated another "faster" RTL description and then performed logic synthesis. The delay constraints were satisfied on the next day. Furthermore, both area and the CPU time for logic synthesis were reduced to 60 and 10%, respectively, on the fourth day. This is because the HLS generated a "faster" RTL structure and, therefore, the logic synthesis tool did not have to perform some delay-optimization tasks such as buffering or logic duplications. This drastically cuts down on area and run time. In other words, controlling delay at the RTL is more difficult for designers and tools than at the behavior level.

In an RT-based design, designers do not like to modify the RTL description once their descriptions are verified by RTL simulation, since RTL simulation takes long time and modification might introduce new bugs. Therefore, in order to synthesize a fast circuit they tend to run their logic synthesis many times under different delay constraints for each module or different synthesis options. HLS users, on the other hand, can easily reduce the delay by changing the place of the clock boundary. The Cyber saves tags between BDL and RTL-HDL, so it inserts the line numbers of the corresponding statements of BDL source code in the generated RTL-HDL (and in a cycle model), and also generates a table for an approximate register-and-register path delay with the corresponding line number of the BDL code. Such information allows designers to find out the portion of the BDL code that needs to be modified. Such modification is a big burden for an RTL designer since the RTL description may be greatly changed, but it is not for behavior description. For such complicated controllers, the logic synthesis toll cannot eliminate serious delay violation even with a long run under various delay constraints and synthesis options, but the HLS has much possibility to optimize the delay by changing its synthesis parameters.

B. Data Intensive Circuits

A data-intensive circuit is more suitable for an automatic-scheduling mode than a control-dominated circuit because its data communication protocol usually has a fixed sequence. Communication is also not frequent, meaning that the scheduler has a lot of freedom to explore various architectures.

1) Huge Reduction of Project Duration: As a typical example of a data-intensive circuit, we introduce our design experience with a digital video-signal processor for a CCD camera with several million pixels in Fig. 4. This video processor was designed with both an RT-based logic-synthesis-oriented design and a C-based HLS-oriented design. The result shows that the C-based design significantly reduces the total project duration. The RT-based design took ten man-months to design this chip, while the C-based design took only 1.5 man-months to design the chip (on an FPGA) and on-board system connected to the camera. (The chip design took less than one month.) The more interesting fact is not that the RT-based designer optimized the chip's area and performance, but that the C-based designer optimized the output picture's quality, as well as area and performance, by modifying the graphic algorithm in his behavioral C description. Such modifications were performed until just before the deadline since the C description is automatically implemented into FPGAs within a few hours as described in the next section. Modifying the hardware algorithm at the RT level, on the other hand, takes a very long time and is very difficult to do without introducing new bugs. Note that this chip is described as one process at the behavior level and its entire behavior is synthesized at once. On the other hand, each module in Fig. 4 is designed and synthesized one by one for RT-based structural design. The synthesis of a behavioral module larger than a structural RTL module is also a key for reducing the design duration using HLS.

2) Real Rapid Prototyping: Although a designer can generate a gate-level netlist from the behavior C description with a HLS system and a logic synthesis, it takes some time to get a satisfactory netlist by adjusting synthesis parameters or behavior descriptions and, therefore, they should perform RTL and gate simulation for timing verification. However, once a good net list is obtained from the C description, it is easy to synthesize it with the same design parameters by slightly modifying the behavior description. At this phase, they can skip RTL and gate-level simulations and directly verify their circuit on FPGA.

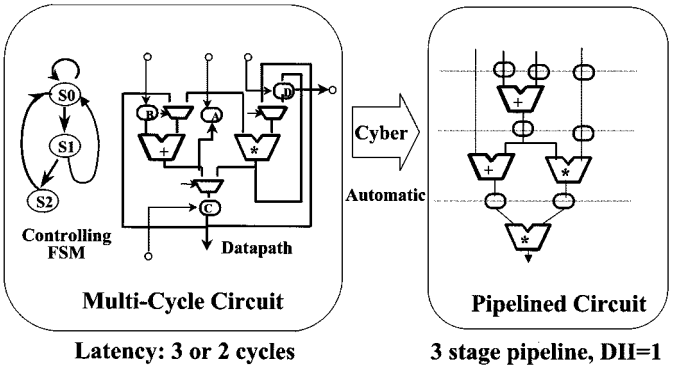

Fig. 5. Automatic pipelining for conditional behavior.

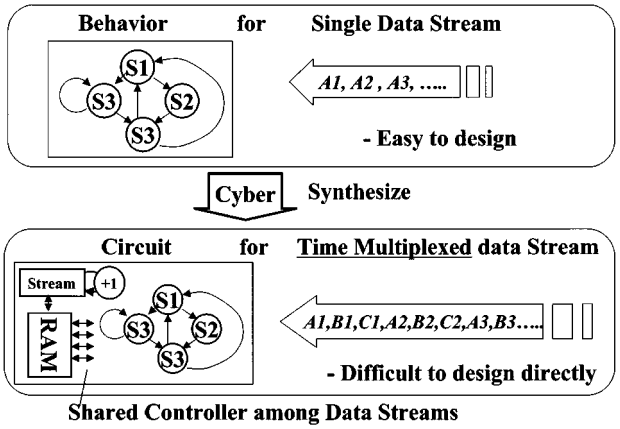

Fig. 6. Automatic generation of TDMA circuit.

For this video processor, the generated circuit on an FPGA had the required level of performance necessary for real video-signal processing. The turn-around time from a behavior to the FPGA is 10 s for HLS, 10 min for logic synthesis, and 30 min for the FPGA layout. Therefore, the designer could debug the chip with real video signals and when they found bugs, modify the behavior, then emulate the modified system again in less than 1 h in the real environment. The problem in FPGA emulation is that it is difficult to observe and control the registers inside the FPGA, such as the state registers. Our synthesizer has synthesis functions that increase the observability and controllability such as: extra outputs are generated which are not necessary but can be used to observe the designated registers during emulation; and conditional breaks are inserted according the registers' values. time when the designated output pins are idling. Some bugs can be found only with emulation under real circumstances, but cannot be found by software simulation. Since our rapid prototyping environment with our HLS enables a designer to debug their circuit at the behavior C level directly after emulation, it reduces the verification period significantly and improves the quality of the circuit.

3) Automatic Pipelining vs. Manual Pipelining: The sequential behavior for the video processor is automatically transformed into a pipelined circuit that inputs video data every cycle and has 15 pipeline stages. This automatic pipelining sometimes surpasses the human designer's pipelining in terms of area. It is complicated for a designer to design a pipeline circuit for conditional behaviors (Fig. 5) since conditional branches require different latencies. according to input data and resource sharing. Therefore, designers usually do not share registers or function units too much in order to avoid bugs; in other words, they sacrifice area in order to achieve comprehensive design. HLS does the sharing thoroughly, and attains smaller circuits. During a Viterbi decoder design trial, our HLS tool surprisingly generated a circuit that was one-tenth the size commercial circuits developed by human designers. Although this may be a special case, HLS tends to generate circuits for complex sequential behavior that are smaller than those designed by human designers.

C. Application-Specific Synthesis and System-Level Synthesis

There are some domain-specific architecture techniques that are not familiar to other domain designers. It is beneficial to incorporate such architectural knowledge into an HLS tool. Fig. 6 shows an example such type of special synthesis function for an SDH network controller [8]. Describing the behavior for a time-division-multiple-access (TDMA) data stream of multiple channels is complicated, but describing the behavior for a data stream of a single channel is fairly easy. Thus, our HLS tool can automatically synthesize a TDMA circuit from the behavior of a single channel's data stream. The number of channels should be input as a synthesis parameter. Our HLS generates a controller for TDMA data processing with one FSM with a memory saving active states of each controller for each channel. Each channel requires one FSM for control, therefore, multiple controls are realized one FSM with the memory. This means multiple controllers are "shared" in one FSM. It is not easy for a nonexpert designer to think of such an architecture or to describe the entire behavior. We think accumulating such domain-specific architecture knowledge leads to a sort of system-level synthesis.

D. Behavioral IP

Design reuse is obviously an important issue for improving not only design productivity, but also design from the higher level abstraction, as discussed above. Peripheral modules connected to a bus, such as a PCI interface, are often reused in our design since their cycle behavior and functions are specified in detail as a standard, even though all functions might not be necessary, or some functions might be missing in commercial RTL/gate IPs. However, RTL IPs are difficult to use for modules which require various performances (throughput) or functions with nonstandard algorithms. This is because modules cannot be parameterized for global performance and functions, for example, necessary processing cycles cannot be adjusted for RTL IPs. Another drawback of RTL IPs is interfacing (to and between them). Designers have to design modules that can communicate with an RTL IP according to the IPs interface specification. When they connect two different IPs, they have to insert an extra interface circuitry in order to synchronize them. Area and delay overhead for this circuitry can not be neglected in some cases.

To provide IPs with more reusability and flexibility, we are developing behavioral reusable components called "behavioral IPs." DES encryption is an good example which illustrates the flexibility and reusability of the behavioral IPs. The behavior description of DES encryption contains a loop of 16 times in the body, and our HLS system can synthesize various performance

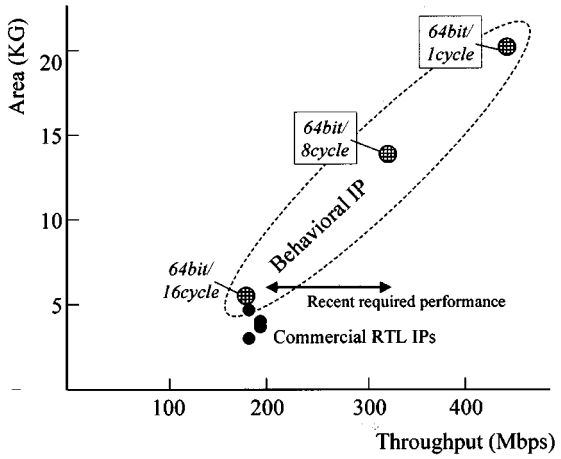

Fig. 7. Behavioral IP for DES encryption.

circuits with various sizes by changing the number of unrolling times. A behavioral IP for DES encryption is provided as an extended C (BDL: which is extended to have parameter variables, such as bit width) function library, so it can just be used as a "functional call" in the C behavior description. Changing the number of loop unrolling times enables the DES encryption circuit to have various cycle performances; 1- to 16-cycles to execute the loop of encryption.

Therefore, the communication sequence between the DES encryption and the caller circuits should be changed according to the DES implementation. The HLS system generates an appropriate interface automatically. (Functions are implemented with inline expansion, as a sequential component or as a sub-routine call in our HLS systems.) It is easier for a designer to use such a function call than a "sequential component" in the behavior since the designer does not have to pay attention to the communication interface protocols. We applied one DES encryption behavioral IP for a variety of chips, which require various performances (157, 288, and 314 Mb/s) or various encryption modes (EBC and CBC modes, as shown in Fig. 7). On the other hand, IPs for 200-Mb/s encryption cannot be used on RTL for 314-Mb/s encryption. Such numbers for unrolled times and encryption modes can be parameterized in a behavioral IP. Therefore, just by setting appropriate parameters, unnecessary functions and redundant interfaces are eliminated in the HLS system, so the reuse overhead for behavior IPs is smaller than that for RTL IPs. Moreover, behavioral IPs can share their hardware resources with other circuits if they are used exclusively, while RTL components basically cannot share resources with other circuits.

E. Interface Synthesis

We presented the synthesis and optimization of an interface for behavioral IPs in the previous section. This section discusses the interface synthesis between processes (behavioral modules).

Communication among processes is often conducted by shared registers or shared memories. The typical interface configuration contains multiplexers with enable signals or address decoders.

Communication through busses between CPUs and hardware modules is created typically as a "memory-mapped input-output (I/O)," where shared registers or I/O devices are allocated an certain address space and a CPU can read them as its memory access. From hardware modules, such shared registers can be accessed as usual components, and hardware modules often send interrupt or trap signal to CPUs. This interface is automatically generated from a certain communication definition table by our HLS system.

Although such a simple combinational interface can be generated automatically just by specifying some parameters or tables, more complicated interfaces such as sequential interfaces (e.g, DRAM bust transfer mode) have to be synthesized from their behavior description. Designers have to describe an interface behavior instead of just specifying parameters. In our HLS system, such an interface function is described as a sequential machine, where clock boundaries are inserted into the behavior. The synthesized controller obeys the interface sequence since the interface function is inserted into the behavior. The interface function is handled as a kind of scheduling constraint, not as an individual component, so it is melted into the entire controller and the penalty of area and performance are minimized.

Currently, our extended C language "BDL" only has a simple communication mechanism through global variables and in/out ports as an interface among processes, and does not have an abstract communication mechanism like channels in the SystemC language [22]. Therefore, a general virtual interface mechanism is not necessary for now because of the simplicity of BDL language. Instead, we are now trying to provide interface function templates for a variety of typical communication protocols. A designer selects or modifies such a template function according to the protocol specification. The categorization of such templates will lead to an intelligent interface synthesis.

F. Discussion

1) Analysis of Synthesized Circuits and Integration with Downstream Tools: Our HLS system adopted a simple target architecture: one process contains one (or multiple) FSM(s) as a controller and a data path, except for the case of application-specific synthesis. Multiple processes communicate through in/out ports or shared registers/memories. Components in a data path are function units, multiplexers and registers, memories, and register files. Function units could be general arithmetic operations and user-defined sequential or combinational circuits described either in BDL, C functions, RTL, or gate level description. The architecture is simple, but an HLS system might generate a large circuit as a single process, which sometimes has a deep sequence. Such circuits are not commonly used in the RT-based design, where combinational modules or modules with a shallow control sequence are executed in parallel.

Therefore, when a HLS system simply generates an RTL description, this can cause some difficulties for downstream EDA tools. For instance, state-decoded signals may have many fanouts since they control all registers, multiplexers, and function units that operate on the corresponding state. Such an interconnect might become a huge buffer tree and cause a timing problem for logic and layout synthesis. To avoid such a harmful interconnect, our HLS system changes the control logic to reduce the number of fanouts of the state-decoded signals.

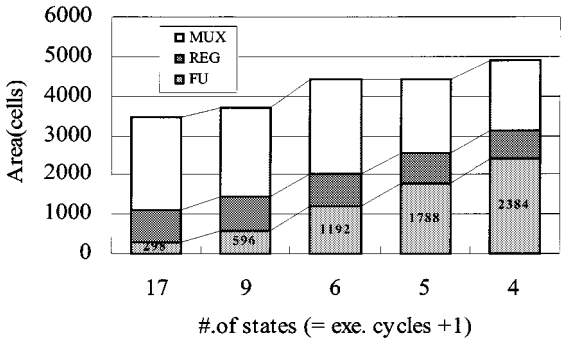

Fig. 8. Area vs. performance trade-off.

Another example is a multi-way multiplexer. Circuits generated by a HLS system sometimes require a huge multiplexer which should be decomposed into smaller multiplexers by an HLS system in order to speed-up the logic synthesis under delay constraints. This modification is actually Boolean optimization, but this kind of optimization should be performed by an HLS system, since it is almost impossible for logic synthesis tools to perform the same optimization for a large circuit without behavior level information.

Fig. 8 shows a trade-off curve between area and performance for a simple graphic processor that contains four different types of filters. The circuit is synthesized under a basic mode where registers will not be clustered into a register file and where the least number of registers is used. The number of multiplexers then becomes relatively large. In such a mode, it is natural that the longer sequence circuit uses more multiplexers and less function units. Information on the composition ratio for function units, multiplexers, and registers can serve as a good guideline for the floor planner. We try to inform downstream tools of such information, but some of it cannot be passed to commercial tools directly, so we try to find a way to pass it indirectly, such as by modifying the module structure or inserting some effective programs into the RTL description. We are developing a new floor planner that can utilize such information.

2) Design Closure: A common problem for EDA tools is design closure. Designers like to approach the target monotonically by changing some tool parameters. Nevertheless, HLS tools generate a very different architecture through minor changes of behavior or synthesis options. For example, partial loop unrolling and complete unrolling produce completely different architectures when all array indexes in the loop become constants after complete unrolling. The indexed array is implemented as individual registers and no address generator is necessary for a complete unrolled loop, but it is implemented as a memory or a register file with an address generator for a partial unrolled loop. A lack of optimization power also causes a significant change in the resulting circuit. For example, exclusive relations among the array index will not be extracted by minor behavior modification. This changes the scheduling. Another major problem is the next HLS version. The new optimization algorithm sometimes optimizes a certain circuit too much, and the generated circuit becomes very different. Unfortunately, unlike the older version, sometimes the new HLS version can not generate the same circuit with options. Currently, some designers save the older HLS version system with the synthesized circuit in case the circuit needs to be modified in the future.

The variety of architecture generation is a double-edged sword for HLS systems. It is important to explore a wide design space, but it makes design closure difficult. Currently, designers replace some initial global synthesis options with some local synthesis programs in the behavioral description, which affects only the related parts of the behavior. Such local synthesis control leads us steadily and gradually toward our goal. In the future these local optimizations will have to be performed automatically under some global design constraints for an entire module, even though this will be extremely difficult since the search space for transformations is too big to examine. Currently, it is difficult to just create local constraints for local codes from global constraints. We are trying to summarize techniques for behavior description and the use of synthesis options based on various actual experiences during chip design. We are hoping that this will provide us with hints for creating a meta search engine for our HLS system.

Until now designs with manual scheduling mode have had no design closure problems. This is mainly because the manual scheduling mode is used to control dominated circuits, which are usually described at a slightly lower level and have fewer function units. In any case, this design closure problem makes it difficult to use an HLS tool extensively for various design areas. We hope to solve this problem by improving the optimization power of our HLS tool.

3) Promotion History of C-Based Synthesis: Since 1986 we have been developing the Cyber High-Level Synthesis system, and in 1994 our HLS system successfully designed a network controller chip as the first type of controller chip. We hoped our success with designing an actual chip would convince many designers to use it. However, for several years the HLS system and its behavioral design methodology were not widely accepted at our company. Although one reason was that our system lacked some synthesis functionalities for designing various types of circuits, the main reason was that designers were afraid to try our experimental HLS system since commercial chip design is always under a tight and strict deadline. Another reason was they could not believe their functional design could be automated. Some of the questionnaires from designers in our company said that “any HLS tool from C must be a toy!” Therefore, to prove to our designers that our current HLS system reached a practical level for industrial designs, our EDA team started to design users’ circuits using our HLS to demonstrate how our C-based design reduces the design duration. They accepted specification documents in Japanese and designed modules in C, and synthesized and verified them in our C-based design environment for a much shorter (one forth or so) period than usual RTL-based design. In addition, we started providing an education course on HLS methodology. With this experience, we learned that a considerable number of hardware designers did not know the C language since they did not perform algorithmic modeling, and the way of algorithmic thinking itself was difficult for them since they were accustomed only to a structural way of thinking in RTL-HDL. It was, interestingly, rather easy to educate software designers to design hardware with the HLS

system. However, some of them described just a logically correct behavior, so we had to educate them that tuning-up the behavioral C code is essential for creating a fast or small hardware, which is also important for embedded software design. Improving the Quality-of-Result of the synthesized chip was of course the most important in promoting the HLS to our designers. Fortunately, the QoR of the synthesized chips has a rather better performance and smaller number of bugs than similar chips designed by RTL-HDL. The small amount of bugs as well as shorter design period also attracted the designers. In addition, we had to examine whether our target architecture, a FSM with a data path, was really general enough for a variety of hardware domains or not. Because of these activities, a considerable number of designers finally began to use our HLS system around 1998. We hope that more than half of the designers in our company will use the HLS system within a decade.

IV. High-Level Verification

This section describes our target verification environment constructed around the Cyber HLS system. Our verification system is based on the assumption that designers use the HLS system and, therefore, each verification tool is tightly tied to the HLS system.

A. High-Level Simulation

1) Significance of the Entire System Simulation: For conventional hardware verification, we mainly rely on RTL-HDL simulation extensively from each module verification to the entire system verification. Unfortunately, RTL-HDL simulation requires too much run time and memory space for million-gate circuits. Even an MPEG2 hardware decoder took four days to simulate the decoding of one standard picture in RTL verilog. System simulation takes several weeks with RT simulation. Accordingly, we have used our hardware RTL simulator [11] for general-purpose computer system design, or a commercial emulation system for processor design. However, constructing such a hardware verification system takes a long period and costs a lot. Consequently, we began to use a C-based behavioral-level simulation in order to simulate a large system including many chips within a reasonable period. Since we perform both behavioral system simulation and RTL simulation for each chip parallely, we have been able to reduce the project duration by increasing the manpower and running the design and verification tasks in parallel more than we did before. For example, a huge system including 20 M gates and 30-MB RAM was simulated in a few hours by the system-level simulation. The fact that the design and verification was finished in just three or four months was due in the most part to such system simulation enable easy description.

2) C-Based Fast HW/SW Co-Simulation and Software Debugging: We have been developing a C++-based software simulator, called "ClassMate," for the entire SoC simulation [10]. Conventional commercial co-simulators use verilog or VHDL simulation for the hardware component, which makes the CPU model run fairly fast, but the entire SoC simulation performance is not fast enough for software debugging. Therefore, in our simulator, all models are described in C++ models for fast simulation. Fast C++ hardware models with accurate cycle behavior are automatically generated by the Cyber HLS system, as will be discussed in the following section. The simulator has various C++ class libraries for CPU models, memory models, abstract bus models, and so on. These models and user-defined models are easily connected through an abstract bus model without detailed communication protocol description in order to construct an entire system model containing many hardware modules and CPUs. Each class has some debugging functions (e.g., break, trace, dump, and scroll), and a GUI interface.

TABLE I
System Simulation Speed in ClassMate

SoC	Embedded Proc.	Sim. Speed*	vs. HLS sim
Audio DSP	original DSP x 2	54 K c/s	1,490 X
HDTV	V850 x 2	23 K c/s	10,000 X
DVD-ROM Decoder	original CPU	52 K c/s	1,750 X

*: Performance on Pentium 450 MHz

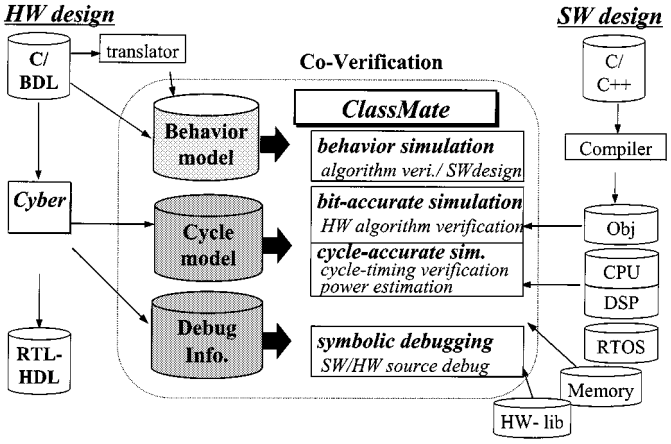

Fig. 9. SOC co-verification flow with HLS.

Our models for CPUs such as V850 and VR4120 and DSPs such as SPX (NEC original DSP) run 300–400K cycle per second with full debugging features. The simulation performance for the entire SoC is shown in Table I. The simulator runs 10- to 10 000-times faster than RTL-HDL-based co-simulators.

3) Stepwise Co-Verification with the HLS System: Fig. 9 shows our SoC co-verification flow with the Cyber HLS system. In the proposed co-verification flow, the designer should perform several abstraction model simulations from the behavior to the RT level. At each simulation phase, different type of bugs are verified. By separating verification items, we aim to reduce the total verification period. Initially, a designer simulates a behavior model, which is the input for the HLS system. Second, they simulate a "cycle-accurate" model which is automatically generated from the HLS system. During the cycle-accurate simulation, our simulator allows the designer to perform debugging on the source code of the input behavior description even though the simulator runs the cycle-accurate model. Embedded software programs on embedded CPUs can be co-simulated with such hardware models at each level. Finally, they perform HDL RTL simulation for verifying remaining problems such as asynchronous communications.

a) Behavior level simulation: The level of abstraction of the behavior model is almost the same as the pure software algorithm, therefore, this model can run 100- or 1000-times faster than the RTL HDL model. For instance, a graphic filter took 4 h for verilog RTL simulation, but only 30 s for bit-accurate C model simulation. Since we usually have to repeat such a simulation many times, this fast simulation will greatly decrease the amount of time needed to finish the project. Also, this model is good for embedded software verification on SoCs because of the high simulation performance. RTL HDL simulation is too slow to debug the embedded software on SoCs.

For the behavior level, we have two types of simulation models. One model is a pure behavior model, which is obtained by compilation with a general C compiler with predefined macros and the executable code is not bit accurate. This model is used for algorithm verification. The other model is a "bit accurate" C++ model, which is translated from the input extended C language BDL. Of course, this model runs several or 10-times slower than the pure behavior model. This model is used for bit-accurate verification, such as quality of graphics.

In addition, since behavior models for hardware modules are available at the early stage of SoC design, software verification can be started with the behavior model. This concurrent design of hardware and software reduces the design period very much, since in the past, software verification had to wait for the completion of RTL HDL design.

b) Cycle-accurate level simulation: Since the algorithm is verified at the behavior level, other problems such as cycle behavior problems (e.g., communication between modules) are verified with the cycle-accurate C++ simulation model. Although the cycle-accurate model has the same cycle behavior as the RTL-HDL model, it runs 10-times faster (or more) than a structural RTL-HDL model. This is because our cycle accurate model is not in a structural RT level, but in a much higher level: a form of state transition. Although all statements in the structural RT model are evaluated every cycle, only statements in an executed state are evaluated in our model. To speed up the SoC simulation, we integrate an automatic abstraction function in our cycle accurate C++ model generator. Although our system level co-simulation adopts an abstract bus communication for fast simulation, the behavior of hardware module has detailed IO behavior for bus communication. Therefore, the detailed bus model is automatically transformed into an abstracted bus communication model based on information from the Cyber HLS system.

Currently, our cycle-accurate simulator is a purely cycle-based simulator, so detailed delay problems, such as an asynchronous communication problem between modules synchronized by different clocks or delay violations in one cycle, have to be verified by the RTL-HDL simulation.

4) Symbolic debugging in behavior and cycle level: Conventional symbolic debugger for C programs can be applied for symbolic source code debugging for behavior level simulation. On the other hand, we need a special tool for symbolic source code debugging for cycle accurate simulation. This is because the structure of a cycle-accurate model is completely different from the input behavior since the HLS applies various kinds of transformations and optimizations, such as: scheduling, resource sharing, loop unrolling and speculative execution. Therefore, it is not easy to locate the part of the behavior that corresponds to the part of the cycle model when simulating the cycle model. Our simulation environment provides designers with a source-level debugging function shown in Fig. 10. The Cyber HLS system generates tags between behavior statements and cycle model statements (of course this is not a simple "one to one" relation, but a "many to many" relation), binds information between variables (arrays) in the behavior, and registers (memories). By using this information, our simulator can indicate the executed statements by highlighting them [Fig. 10(a)] and the value of variables and the bound register names [Fig. 10(b)] in the behavior source code in an active state of the cycle simulation model. In other words, a designer can debug their design directly on the behavior source code when cycle-accurate simulation is performed. With this symbolic debugging function, designers do not have to look at the generated cycle-accurate C++ model at all, which is usually difficult to check. The tag and binding information between the behavior and the cycle model are used for symbolic debugging also for the RTL-HDL simulation. In addition, the information can be used for creating a symbolic debugger for a gate-level simulation although we have not implemented it yet.

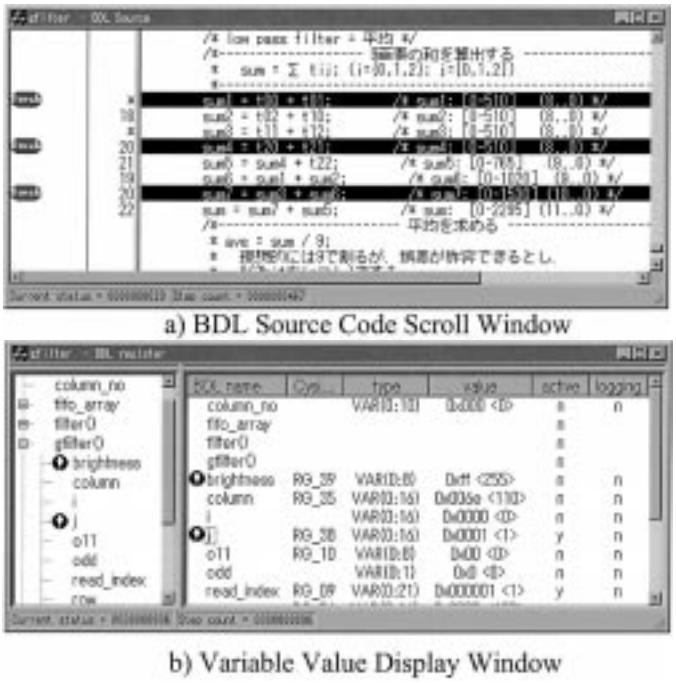

Fig. 10. BDL symbolic debugger.

Finally, in order to estimate the power consumption of the synthesized hardware, power calculation statements are automatically inserted into the RTL-HDL and cycle models. This power estimation is much more accurate than conventional power estimation for manual RTL-HDL, since it is customized to the target architecture of our HLS system.

B. Formal and Semi-Formal Verification

1) Formal Verification with HLS: Since 1993 we have been very successful with our use of the formal equivalence checker

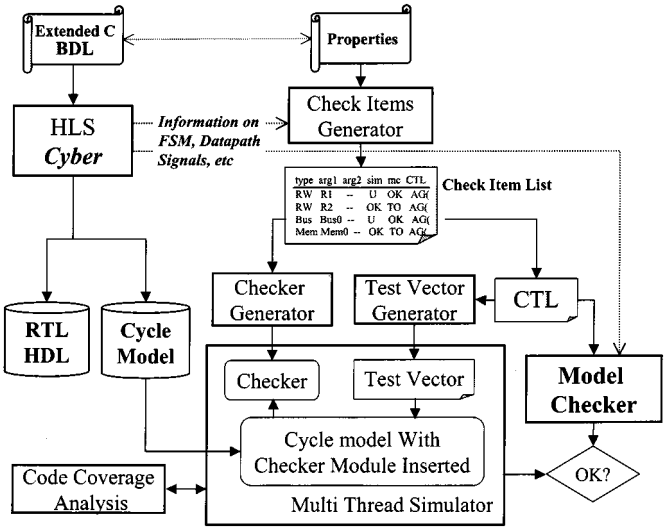

Fig. 11. Formal and semi-formal verification.

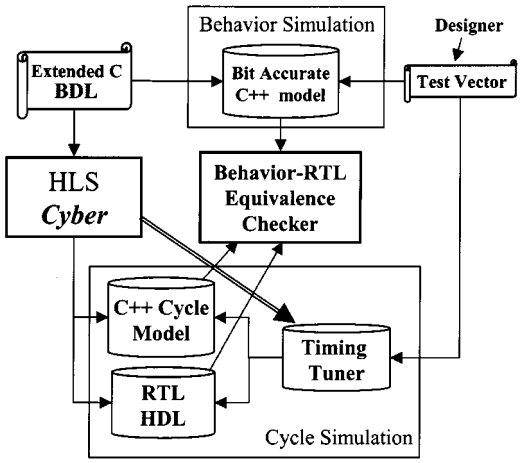

Fig. 12. RTL testbench generator and equivalence checker between RTL and behavior.

between the RTL and gate-level descriptions. Therefore, quite a few designers have requested an RTL property checker or a high-level formal verification tool. We have been developing an RTL property checker with a BDD (Binary Decision Diagram)-based symbolic enumeration state traversal technique [17] and with an SAT engine [18]. However, our current property checker allows us to verify only small circuits designed by manual RTL design. As a first step, therefore, our formal property checker aims to verify only the RTL description for one process synthesized by the Cyber HLS system. An entire SoC containing multiple processes will be verified by simulation-based techniques, as presented later. Our formal verifier takes advantage of various information on the generated RTL description from the HLS system, such as partitioning of a controller and datapath, state transition, register sharing, and register information. The information is almost impossible for current techniques to extract from a manual RTL description. It is available in order to handle realistically-sized circuits in model abstraction and narrowing state traversal space, etc. As a preliminary experimental result, our property checker was able to verify an RT circuit in 15 s, however, the VIS system [15] could not verify this since it went over the time limit.

2) Formal Verification Environment for SoC: In order to perform an entire SoC verification, we are developing an integrated verification environment for a simulation-based verifier and a property checker, as shown in Fig. 11. This verification is a complementary phase to the co-simulation discussed in the previous section. A designer can use both the model checker (formal property checker) and simulation-based verifier via the same user interface. They specify, in a comprehensive form, some properties which their design must satisfy. The properties are translated into a checklist item, which is used for both the model checker and the simulation-based verifier. For the model checker, a CTL (Computational Tree Logic) formula [16] is automatically generated from the list. For the simulation-based verifier, checkers such as diagnostic statements for illegal conditions, are generated from the list, and the checkers are inserted into the cycle-simulation model generated by the Cyber HLS system. A testbench to activate such diagnostic statements is also generated by using the CTL formula. A designer knows the code coverage rate: i.e., how many statements are verified with the given test vectors.

3) RTL Testbench Generation and Equivalence Check: One of the advantages of C language simulation, when compared with HDL simulation, is that it contains very useful libraries which can generate test vectors for the design model. For example, C library functions like a random number generator, mathematical library and signal-processing library. With these models, a designer can generate pseudotest vectors similar to the vectors in the actual usage environment. Naturally, designers hope to use the test vectors for RTL or cycle simulation. However, the behavioral test vectors cannot be used in RTL or cycle simulation as is unless all the communication interfaces in the behavior are in hand-shaking style. We, therefore, need a timing tuner model which enables the RTL or cycle model to input the test vectors into the correct cycle. The timing tuner is created based on which data are read and written (and when they are read and written) from the HLS system. We have found during the industrial design of chips that a behavioral test vector with a high code-coverage rate also has a high code-coverage rate for RTL simulation. Finding high-coverage stimulus in the behavior level is easier for designers than finding it in the RTL level. In addition, this capability makes it possible to check whether the behavior and RTL model are equivalent by comparing the output transactions of both level simulations. The equivalence checking is important since we should always pay attention to bugs in the HLS system (see Fig. 12).

C. Effect of C-Based SoC Design Environment

In this section we show how our C-based SoC design environment will reduce the time needed for the SoC project by analyzing our design project of an SoC for mobile communication.

We compared the design duration with the conventional RTL design method and the proposed C-based design method, as shown in Table II. Note that gate size for C-based design is three-times larger than that for RTL-based design and some numbers for C-based are estimated value on preevaluations. Design processes other than those described in the table are: layout,

TABLE II
EFFECT OF C-BASED SOC DESIGN

$0.25\mu m$	#gates	before	after	runtime
data1	0.8M	20.83ns	9.80ns	3h30m
data2	3.5M	64.70ns	15.53ns	7h20m

chip sample production, hardware evaluation and software evaluation. In Table II, "project duration" indicates the total project period for all design processes (including the above processes not in the table) which is not the sum of the necessary man-month of all processes, since some design processes are performed concurrently. In the C-based method, designers perform system-level C simulation in the specification phase. The C description is used for the HLS process. The effect of the C-based method is effective for "hardware logic design" and "embedded software design." The design period for both is drastically reduced: The hardware logic design period is reduced from 7 mo to 1 mo, and the software design period is reduced from 5 to 2 mo, even for the larger-sized SoC. The total project duration is also decreased from 14 to 8 mo.

V. EXTENDED C LANGUAGE: BDL

Our HLS system accepts behavioral VHDL, Verilog, and Java in addition to the extended C language "BDL." Several years ago, many designers liked to use behavioral VHDL or Verilog. However, recently they shifted their SoC design environments into C or C++ language. This is because the embedded software is described in C and the simulation speed and design environment of C is better and more reasonable than HDL. The reason why we selected C but not C++ and Java, is that the BDL language originally was designed in 1987 when C++ was not so popular and Java had not yet emerged. Technically, C++ with hardware class libraries such as SystemC or Cynlib [23] is superior to C in the sense that the semantics of C++ do not have to be extended explicitly to describe some behavior particular to the hardware although those of C do have to be extended. We plan to support library-based C++ language as the input language of our HLS system. Currently, we do not have a plan to support a new system-level HDL, such as SLDL [20], superlog [21] or object oriented VHDL (OO-VHDL) although they could be very good system-level languages, since we would like to use a single language for hardware and software design.

A. Extensions and Restrictions

Three types of **extensions** to the C language are necessary to design a realistic hardware description language. The first type is made up of inevitable extensions that describe the behaviors particular to the hardware. The second type is made of extensions that concisely describe hardware behavior. The third type is made of the behaviors that control the quality of the synthesized circuit. The usage of the second and last types of extensions depends on the designers preference. Though all the types are incorporated in the BDL language, the Cyber HLS system basically does not require the second and the third to synthesize circuits. These extensions enable designers to describe even an RTL behavior in BDL, and to tune-up the code as in details as they want.

For the first type of extension, BDL has "input and output port" declarations and "bit-width" of variables for a general circuit, and "synchronization" [with wait() function], "clocking" (with clock-boundary symbol), "concurrency" (with multiple processes), and "interrupt/trap" (with always construct meaning statements in the body are evaluated every cycle). For the second type of extension, BDL has hardware-oriented operations, such as "Bit extraction," "reduction and/or," and structural components, such as a multiplexer. In the last type, BDL has hardware-oriented operations, such as "add-with-carry operation of three inputs," and "decoder" to save area. It also has indicators for hardware implementations, such as, variables implementation (e.g., registers or wires for variables and memories or register files for arrays) and data-transfer types for variable assignment (e.g., data transfer through wire, registers, latches, or tri-state gates). Many programs (synthesis and/or simulation directives in a comment) are also thought of as the third extensions. Examples of synthesis programs include: synchronous/asynchronous sets of initial register values, specification of gated clocked register, partial-loop-unrolling number, and manual binding indicators. An example of simulation programs is an indicator to variables connected to CPU bus, which is used to make an abstract bus model for co-simulation as discussed before. Since the BDL can express cycle-accurate behavior, it is suitable to describe a test bench. For describing a test bench, nonsynthesizable descriptions, such as system calls or library functions, can be used in BDL. Diagnostic modules observing the outputs of designs can also be similarly described. Recently a few special languages for testbench generation have been proposed for test bench description such "VERA" [25] or "e" [26]. They are powerful, but we think it is not easy to learn a new language and to use two different languages for testbench and design description. We believe that so far one design language for SoC, including testbench design for hardware and software, might be better.

On the other hand, BDL has some **restrictions** for some semantics of the C language. The BDL supports almost all control structures such as: loops, conditionals, go to, break/continue, functions, and side effects. The go to statement is important in the control dominated circuits design since the circuits require exceptional jumps to any control state with only one cycle. Data types such as multi-dimensional arrays, struct, static variables are also supported. However, currently, dynamic behavior such as dynamic allocation with pointers, and recursion are deleted as a "synthesizable" subset since we cannot estimate their necessary memory size during synthesizing, when we implemented them in a linear memory. When a designer newly describes any hardware behavior in C, these restrictions are no problem at all. However, when we try to synthesize from Legacy C codes, such restrictions are big burden for the designer to rephrase them. The pointer analysis-based synthesis approach [19] tries to loosen such restrictions. We think supporting pointers might be very important for a compiler of dynamic reconfigurable chip as will discussed later.

B. Discussions on Using C as a Hardware Language

The difficulties designers face when designing some domain chips are often due to the inferiority of the C language as a

hardware description language. C language is suitable for algorithmic behavior and BDL is also suitable for state-based specification, but they are not suitable for data flow and continuous computation (Matlab [24]). For example, BDL is not suitable for the handling of stream data in digital-signal processing. Area-specific language, such as Silage, can handle "delay" and capture the flavor of the signal flow graphs in a textual way. Calculation data forwarding, which is often used in the pipeline design in processors, is a similar type of circuitry and is not easily described in C due to the lack of delay. This type of circuit is basically suitable for structure-oriented thinking, but not for behavior-oriented thinking. Currently, designers often describe such circuits in a structural way in BDL, not in a behavioral way, since describing them in a behavioral way is rather hard to conceptualize. In this case, our HLS is thought to be just a language translator from BDL to RTL-HDL, but designers can take advantage of the fast C++ system simulation environment. Some special pipeline architectures which cannot be automatically generated by our HLS system are also described in structural BDL, but this fact is not the problem of BDL since they would be synthesized by enhancing the HLS system in the future.

VI. Link to Physical Design

This section describes the possibility of incorporating physical design features into a C-based SOC design environment.

In DSM design,the interconnect has generally been accepted as the dominant factor in determining the overall circuit performance and complexity [31]. The major problem when applying conventional design flow for 0.5 μm and above processes to DSM design is that most of the timing and area optimization in higher-level design is based on an inaccurate estimation of the interconnect delay and complexity. The difference between the estimation and the actual post-layout value is significant in DSM designs, which results in the failure of design in terms of both timing and area.

One solution to this problem is to integrate logic domain tools and physical domain tools [31]. For gate-level design optimization, logic synthesis and place-and-route (P&R) tools are now being integrated. The next step is to integrate our HLS and our floorplanner for RT-level design optimization.

A. Design Integration at Gate-Level

We have started to incorporate technology-dependent netlist optimization: gate sizing and rule-based repeater insertion (which are conventionally applied in the later stage of logic synthesis) into the physical design system for 0.35 μm process LSI design. Currently, more advanced algorithms that incorporate buffers/inverters insertion, gate sizing, interconnect-topology generation, and placement based on precise physical information are available [32]. By combining these techniques with circuit partitioning and adaptive timing budgeting in the physical design system, which is called *physical optimization*, we have been able to successfully design high-performance chips from 0.35 μm through 0.18 μm.

Table III shows an example of how the worst delay can be improved by the physical optimization of a large DSM design

TABLE III
Worst Delay Improvement by Physical Optimization

Project Duration		HW Logic Des.: 7M		
RTL-based Design	Spec.	RTL+LS	Veri.	SW Des.
14 M for 200KG	3Mx4p	6Mx4p	4Mx2p	5Mx5p
Project Duration		HW Logic Des.: 1 M		
C-based Design	Spec.	HLS+LS	Veri.	SW des.
8 M for 600KG	3Mx5p	2Wx4p	1Mx4p	2Mx5p

(the worst delay "before" and "after" the physical optimization). The "runtime" for the optimization is measured on a Sun UltraSPARC II (450 MHz) with 2.0-GB RAM, and the amount of memory used for 3.5-million-gate LSI is about 1.5 GB. The chip performance required for current advanced designs can not be met without these technologies.

B. Block-Based Design with Floorplanner

Another key methodology for achieving high-performance DSM chip design is using a floorplan for design [33]. A structured block-based design is popular for designing microprocessors or other very high-performance custom designs. In this method, the entire design is partitioned into a set of blocks with 10–100K gates, basically according to logical hierarchy, and the position of the block is determined prior to gate-level logical or physical design.

The main purpose of design planning is to estimate chip performance and routability in the early stage of the design. Timing budgeting for blocks, custom wire load model generation, and interconnect planning is determined based on this analysis, which must take repeater insertion effect into account. For a fully customized chip such as microprocessor, the floorplan is carefully optimized by designers iteratively, but for the application-specific integrated circuit design, whose time-to-market is very crucial, the automatic floorplanner (block placer) is an essential component in block-based design. Fig. 13 shows an example of a floorplan (block placement and shape optimization with preplaced blocks) obtained by our floorplanner for a 3.5-million-gate LSI. It took about 10 min to generate the floorplan and the result is comparable with a floorplan designed by a skillful designer. Fig. 14 shows delay estimation for inter-block wire at the RT and gate levels in our system. The same algorithm was used for repeater insertion in both levels. The RT-level estimation is lower accurate than the gate-level estimation because the former estimation is performed from and to a certain positions of the block boundaries, while the latter estimation is based on the exact position of gates inside the block. These estimates are accurate enough to be used in RT-level design. Thus, these analysis and synthesis technologies for block-based designs can help designers explore design alternatives efficiently in the early stages of design.

C. Physical Design in C-Based SOC Design Environment

The possibilities of integrating logical and physical design at the gate-level design have been investigated so far, and encouraging results have been obtained in, for example: buffering, gate sizing, technology re-mapping, duplication, and redundancy re-

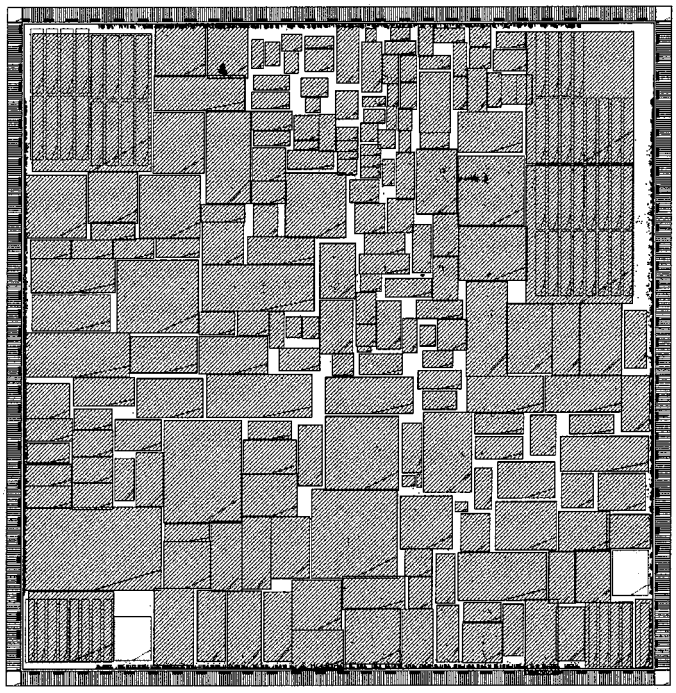

Fig. 13. Floorplan for 3.5-million gates LSI (#Soft blocks: 202, #Hard blocks(preplaced): 59, #Inter block nets: 45 078).

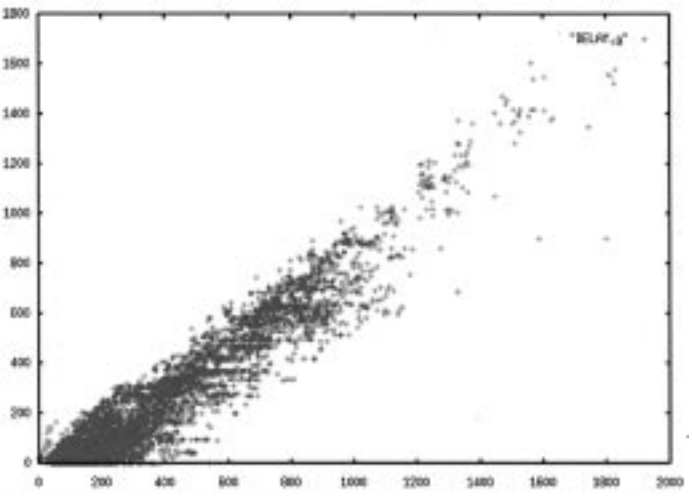

Fig. 14. Delay estimation for inter block wire in RT and gate level.

moval and addition. However, it is not good way to integrate a full set of complex logic optimization technologies into physical design and construct a single-pass design flow from RTL to GDS with flat-design style. Appropriate coupling and de-coupling in the design flow is important, and as a next step we have to look into establishing an efficient block-based design framework. While the possibilities for gate-level design optimization are limited, RT-level design optimization can achieve a higher level of optimization in the block-based design framework.

In the block-based design, we can optimize RT-level design together with design planning for block partitioning, placement, and global interconnect (e.g., topology, wire sizing, and repeater), timing budgeting, which is lacking in most current design styles. If we have an optimized RT-level design with good planning at an early stage in the design, intra-block design can be achieved with conventional logic synthesis technologies focusing on gate-delay, followed by physical design with simple technology-dependent optimization, such as buffer-tree re-construction, gate sizing, and technology re-mapping. Therefore, efficient exploration for design alternatives at the RT-level coupled with the design planning is a key point for designing an SOC with more than ten-million gates.

The integration of the HLS and floorplanner makes this exploration possible. For example, if an interconnect never meets the delay constraint according to floorplanner, the HLS could change the clock boundary to ease the delay constraints for the interconnect. In an RT-based design, the designer has to modify the RTL description by themselves, but the modification is a difficult task and tends to introduce new bugs. HLS users, on the other hand, can easily reduce the delay by changing the place of the clock boundary. Although a serious delay violation cannot be resolved by a logic synthesis tool, even with a long run for various delay constraints and synthesis options, HLS can controlled much more easily by changing the synthesis parameters. Accurate wire-load estimates from the floorplanner would also be helpful for RT-level power optimization. Appropriate HLS parameter generation for behavioral IPs with the floorplanner are necessary for implementing the design (IP) re-use environment.

In order to develop this design environment, we need a methodology that appropriately integrates the HLS and floorplan in order to explore RT-level design space efficiently. For example, how to combine the HLS-driven floorplan and floorplan-driven HLS (Similarly, we discussed synthesis-driven layout and layout-driven synthesis for gate-level integration.) And how to cluster and partition the components generated by HLS to make the circuits suitable for the floorplanner, where logical hierarchy and signal flow, as well as connectivity have to be taken into account for performance optimization.

VII. Design Issues for Next-Generation SoC

In the above sections we discussed the future direction of HLS and verification, SoC language, and physical design. This section describes several of our future topics: design, architecture, EDA tools, and design methodology for the next-generation SoC.

A. New SoC Architecture and EDA tools

1) Simulation of Heterogeneous Components: The current growing demand for embedded systems, such as wireless portable personal data assistants, is pushing forward the integration of heterogeneous components on SoCs, such as: DRAMs, Flash Memories (involatile memories), DSPs, CPUs, signal processing hardware, hardware controllers, bus interfaces, and analog and RF circuits. Currently, the production cost of such SoCs is said to be higher than that of System-on-Board or system-in-package (SIP), but we hope improvements in production technology will solve these problems in the near future. Therefore, a design flow and tools for such heterogeneous SoC chip needs to be created. Firstly, a system simulation of the entire SoC is required. There is some research underway being conducted on digital and analog

co-simulation in C language. These activities might be very suitable for our current C-based design flow. Multi-language co-design environment [28] might be necessary for modeling and implementing such a heterogeneous SoC. Other important remaining issues for such a heterogeneous SoC are: testing, timing analyzer, and reliability checking. Current automatic test pattern generation or boundary scan techniques (JTAG) could be used for testing such chips somehow, but of course could not attain enough testability. The current static timing analyzer (STA) deals with only static CMOS circuits, so we have to create a timing analyzer for this type of SoC chip.

2) SoC with Reconfigurable Computing: Due to the rapid improvement in MPU performance many applications are being implemented as software instead of hardware. Consequently, some people have the opinion that all digital behavior will be implemented on MPUs and that hardware will remain simply as an interface circuit. However, we believe that for at least a few decades demand will remain for high-performance hardware that consumes little power, since new types of processing will emerge that require more and more computing power, such as high-density motion picture processing. Some forms of processing will be able to be executed only by hardware. Even in this case, programmability is key for hardware's survival. Programmable hardware such as a dynamic reconfigurable FPGA will be an important component of the next-generation SoC. Accordingly, we have been developing dynamic reconfigurable logic (DRL) chips [12], [13]. The DRL chip has one FPGA plane and several (16 to 256) configuration RAMs for the FPGA plane. The behavior is "temporally" divided into multi-contexts and each context is saved in a RAM storing the FPGA plane configurations.

Reconfiguration of the FPGA plane, namely context switching, takes only 4.6 ns, so when clock period is more than 4.6 ns this DRL chip can execute the behavior by switching context every even cycle. This technique enables behavior larger than a conventional FPGA to be implemented in a DRL chip. In other words, the DRL chip can become a kind of "virtual-computing" hardware. In addition, the DRL chip has 200- to 1000-times improvement per power improvement than general processors because of its massively parallel register and memory accesses and concurrent spatial processing of fine-grained data. For example, we ran a DES encryption algorithm on FPGA and a Pentium processor. Both the FPGA netlist and Pentium executable code are automatically synthesized from the same C description by our HLS system and by the gcc (Gnu C compiler). Even though the Pentium uses clock frequencies that are ten-times faster than FPGA, the encryption performance of FPGA is several-hundred times better than that of the processor. Such dynamic reconfigurability is useful for various embedded applications such as the Internet packet and ATM packet processing, wireless terminals, image recognition, and picture CODEC. Area specific processors such as network processors, multi-media processors, and specially-designed DSPs might also become popular as a substitute for MPUs. However, for these processors, designers usually have to describe algorithms in assembly code, not in C language, because establishing a good compiler for special instructions is often difficult. In contrast, the DRL compiler can be implemented based on the current HLS technology. This is because the compiler divides the behavior temporally so that context switching occurs as little as possible. It might be more realistic to implement a good DRL compiler than to implement an automatic compiler generator for such application-specific processors. We believe such type of dynamic reconfigurable chip will become an essential component on the future SoC.

3) Silicon RT-OS: Many embedded systems have recently adopted the use of Real-Time Operating Systems (RT-OS). The need for fast and hard real-time response could be fulfilled by silicon RT-OS. In a pioneering work, a silicon RT-OS for i-TRON has been reported [14]. This chip is able to process the most dynamic system calls. It accepts interrupt signals from hardware and performs task scheduling or some real-time behaviors, and sends data to the CPU with a CPU interrupt signal. The chip could be an important IP for an embedded system.

4) Interface between CPU and Hardware: Currently, the way how hardware and CPUs communicate is very limited. Interrupt/trap or shared-mapped IO memories are often used, as previously described The software handshaking offers flexible communication between hardware and software, but it is too slow. This slow latency of communication makes the hardware-software partitioning problem difficult. Developing new communication architecture (or interface) of the CPU that enables fast data communication to the outside might be the key to perform automatic hardware-software partitioning and co-synthesis from system-level algorithmic description. This communication architecture would also be beneficial for the DRL chips and the silicon RT-OS.

5) General Virtual Interface: The concept of a generic virtual interface has been attracting a lot of attention as a way to increase the design reuse or IP business. General virtual interfaces such as Virtual Socket Interface Alliance (VSIA) and Virtual Component eXchange (VCX) lead to designers to believe that any IP could communicate with any other IP. Despite their great efforts, designers still face many challenges, especially considering the communication timing problem. Some practical approaches are reported such as an automatic matching/generation/deletion of interface pins [27]. General virtual interfaces are kinds of wrapper IPs, so they would have the area and delay overhead. It is sometimes difficult for a logic synthesizer to delete or reduce them for synthesizable RTL IPs. We think behavioral IPs have higher potential as we discussed before.

In the area of architecture design, some researchers are trying to develop general communication interfaces in hardware. For example, some are trying to apply telecommunication techniques for interfaces in a gigahertz chip, such as communication with code-division-multiple-access (CDMA) technique on a system bus since the CDMA communication is robust for noise, which might be a big problem in an ultra-high-speed DSM chip.

Regarding circuit design, the asynchronous circuit interface has attracted a bit of attention from designers as a general IP interface because asynchronous communication with "C-elements (Muller's C-elements)" might enable more general and

faster handshaking than synchronous handshaking. Such circuits could also be used for high-performance chips. Since this interface enables arbitrary communication between modules in different clock-frequency domains, the local clock frequency of each module could be much faster than the global clock which is determined on the slowest module [34]. It will be necessary for system-level EDA tools to be able to optimize circuits containing such different clock domains.

6) New Layout Architecture: Since interconnect delay is becoming an increasing problem, many new device technologies, such as copper wiring and low dielectric insulators, are being developed to minimize this delay. Here we summarize the guidelines for various layout rules (i.e., routing, and which layout tools to use when modifying routing). However, modifying the routing paths is not the final solution. We need to propose a better layout architecture for DSM chips.

Another issue is the integration of logic domain tools and physical domain tools. Logic synthesis and place-and-route (P&R) tools both handle gate-level circuits and are now being integrated to meet the DSM delay constraints. Similarly, HLS and floorplanner should be integrated since both handle RT modules. For example, if an interconnect never meets the delay constraint according to our floorplanner, HLS could change the clock boundary to ease the delay constraints for the interconnect. HLS may be required to have the floorplanner incorporated inside it in order to estimate the global interconnect delay, or to communicate dynamically with the floorplanner.

B. Design Methodology

The C-based SoC design methodology is now being accepted in our company, and we hope such methodology will be accepted worldwide. Our goal for the C-based hardware design methodology is to make the productivity of hardware design as high as that of software design. Consequently, it might be a good idea to apply some software design methodology to our C-based SoC design. For the Object Oriented (OO) language design, the notion of "design pattern" [7] and modeling methodologies such as "unified modeling language (UML)" or "Entity-Relationship chart" have become accepted over the last few years. This sort of formalized analysis and modeling for SoC designs might be beneficial for storing and categorizing design and coding techniques for reuse. In particular, the C coding techniques for obtaining smaller and faster chips should be analyzed and categorized. Also, domain-specific architecture synthesis should be implemented for various areas, as discussed in the HLS section. Therefore, we have to develop a way to allow designers to incorporate such architectural knowledge into our HLS or system level synthesis tools by themselves in the form of libraries, although currently our EDA team incorporates such knowledge directly into the synthesis algorithm.

The OO language environment provides a lot of software objects with design patterns indicating how these objects are used for a categorized application. For our SoC design environments, we have to supply reusable behavioral IPs such as encryption, DCT, distance calculation, floating point arithmetic, and graphical filters. These components must be at the behavioral level because their structures can be modified according to hardware constraints and they can share hardware resources. For instance, a floating-point arithmetic unit requires a large area for the combinational component, but it also requires a much smaller area for the sequential component.

However, a C compiler for hardware has to optimize the circuit more thoroughly than that for software. This is mainly because hardware designers are more concerned about the size and performance of the generated chip than software designers are concerned about them of the generated executable code. (As described before, this is not true for embedded software at all.) Software reusable components can be generic since they can contain as many as functions, since the redundant codes for unused functions are not a big problem if they remain in the executable code.

Finally, in a software C compiler, C description is a final implementation level, but in our C-based design environment, designers still have to operate downstream tools such as logic synthesis and layout. Because this process is not fully automatic, hardware-design productivity is much lower than software design productivity. We think that programmable hardware, like DRL, could have the same productivity software, and therefore, incorporating such programmable hardware in SoC is very important from the view point of design productivity.

VIII. Conclusion

In this paper, we described our achievements and vision for our C-based SOC design methodology and EDA tools. For upstream design flow, we discussed the advantages and disadvantages of our HLS and verification flow by analyzing several industrial chip designs. We believe that the C-based design methodology with the HLS system really represents a paradigm shift in SOC design, and that it will overcome the design productivity gap problem. For downstream design flow, we described how to incorporate physical design features into the C-based SOC design environment, which has great importance in DSM VLSI design. The block-based design environment, which integrates the HLS and the floorplanner, will make it possible to explore RTL design space efficiently by taking DSM design effect into account. Finally, we presented several future topics on SoC architecture and design methodology for the next-generation SOC.

Acknowledgment

The authors would like to acknowledge their many colleagues at NEC for their input on various topics presented in this paper. They would also like to thank H. Tanaka, Y. Nakagoshi, K. Hirose, and W. Takahashi for their contribution in implementing our HLS and other tools and H. Kurokawa and H. Ikegami for their discussion on co-simulation. They would like to thank A. Mukaiyama for his discussion on formal verification, K. Nakayama for his suggestions on Signal Integrity problems, A. Kawaguchi for his discussion on Design Pattern and UML modeling, and K. Oyama for contribution on behavior IPs and Fig. 7. Finally, they would like to thank T. Yoshimura for his contribution in developing a prototype of the algorithm and system for our floorplanner.

REFERENCES

[1] *The National Technology Roadmap for Semiconductors: Technology Needs*, 1997 ed. San Mateo, CA: Semiconductor Industry Association, 1997.

[2] D. Gajski *et al.*, *High-Level Synthesis: Introduction to Chip and System Design*. Norwell, MA: Kluwer Academic, 1992.

[3] G. De Mecheli and M. Sami, *Hardware/Software Co-Design*: KAP, 1995.

[4] J. Rozenblit and K. Buchenrieder, *Codesign: Computer Aided Software/Hardware Engineering*. Piscataway, NJ: IEEE Press, 1995.

[5] L. Semeria and A. Ghosh, "Methodology for hardware/software co-verification in C/C++," in *Proc. ASP-DAC 2000*, 2000, pp. 405–408.

[6] E. Gamma, *et al.*, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[7] K. Wakabayashi, "Cyber: High level synthesis system from software into ASIC," in *High-Level VLSI Synthesis*. Norwell, MA: Kluwer Academic, 1991.

[8] K. Wakabayashi *et al.*, "Design transmission equipment with behavior synthesizer," *Nikkei Electron.*, no. 655, 1996.2.12, pp. 147–169, 1996.

[9] K. Wakabayashi, "C-based synthesis experiences with a behavior synthesizer cyber," in *Proc. DATE '99*, 1999, pp. 390–393.

[10] H. Kurokawa, "C++ simulator 'ClassMate' for preverification on SOC," presented at the IEICE, 1998, Paper VLD98-46.

[11] S. Takasaki, N. Nomizu, Y. Hirabayashi, H. Ishikura, M. Kurashita, N. Koike, and T. Nakata, "HAL III: Function level hardware logic simulation system," in *Proc. IEEE Int. Conf. Computer Design*, 1990, pp. 167–170.

[12] M. Motomura *et al.*, "An embedded DRAM-FPGA chip with instantaneous logic reconfiguration," in *Proc. Symp. VLSI Circuits*, 1997, pp. 55–56.

[13] T. Fujii *et al.*, "A dynamic reconfigurable logic engine with a multi-context/multi-mode unified-cell architecture," in *Proc. ISSCC'99*, 1999, pp. 360–361.

[14] Nakano *et al.*, "VLSI chip design for real time OS and its evaluation," in *Proc. 10th Karuizawa Workshop*, 1997, pp. 333–338.

[15] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa, "VIS: A system for verification and synthesis," in *Proc. Computer Aided Verification*, 1996, pp. 428–432.

[16] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach," in *Proc. 10th ACM Symp. Principles of Programming Languages*, 1983, pp. 117–126.

[17] J. R. Burch,, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Sequential circuit verification using symbolic model checking," in *Proc. 27th Design Automation Conf.*, 1990, pp. 46–51.

[18] A. Gupta and P. Ashar, "Integrating a Boolean satisfiability checker and BDD's for combinational verification," in *Proc. VLSI Design 98*, 1998, pp. 222–225.

[19] L. Semeria, *et al.*, "Resolution of dynamic memory allocation and pointers for the behavioral synthesis from C," in *DATE'00*, Mar. 2000.

[20] SLDL [Online]. Available: http://www.inmet.com/SLDL

[21] P. Flake and S. Davidmann, "Superlog, a unified design language for system-on-chip," in *Proc. ASP-DAC 2000*, 2000, pp. 583–586.

[22] SystemC, OSCI [Online]. Available: http://www.systemc.org/

[23] Cynlib [Online]. Available: http://www.cynapps.com

[24] MathWorks (1998). [Online]. Available: http://www.mathworks.com

[25] S. AL-Ashari, "System Verification from the Ground Up," *Integrated Systems Design*, Jan. 1999.

[26] "e" Verisity [Online]. Available: http://www.verisity.com/

[27] R. Bergamaschi, "Automating the drudgery in system-on-chip design," in *Proc. SASIMI 2000*, 2000, pp. 269–275.

[28] P. Coste, *et al.*, "Multilanguage codesign using SDL and Matlab," in *Proc. SASIMI 2000*, 2000, pp. 49–55.

[29] VSI [Online]. Available: http://www.vsi.org/

[30] VCX [Online]. Available: http://www.vcx.org/

[31] K. Keutzer, A. R. Newton, and N. Shenoy, "The future of logic synthesis and physical design in deep-submicron process geometries," in *Proc. Int. Symp. Physical Design*, 1997, pp. 218–224.

[32] T. Okamoto and J. Cong, "Buffered steiner tree construction with wire sizing for interconnect layout optimization," in *Proc. IEEE Int. Conf. Computer-Aided Design*, 1996, pp. 44–49.

[33] D. Sylvester and K. Keutzer, "Getting to the bottom of deep submicron," in *Proc. IEEE Int. Conf. Computer-Aided Design*, 1998, pp. 203–211.

[34] S. Schuster, W. Reohr, P. Cook, D. Heidel, M. Immediato, and K. Jenkins, "Asynchronous interlocked pipelined CMOS circuits operating at 3.3–4.5 MHz," in *Proc. ISSCC'2000*, 2000, p. 292.

Kazutoshi Wakabayashi (M'95) received the B.E. and M.E. degrees from the University of Tokyo, Tokyo, Japan, in 1984 and 1986, respectively.

He joined NEC Corporation, Kawasaki Japan, in 1986 and is currently a Research Manager of the Computer & Communication Media Research Laboratories. He was a Visiting Scientist at The Center for Integrated Systems, Stanford University, Stanford, CA, during 1993 and 1994. He has been engaged in the research and development of VLSI CAD systems; high-level and logic synthesis, formal and semi-formal verification, system-level simulation, HDL, emulation, HLS and floorplan links, and reconfigurable computing.

Dr. Wakabayashi has served on the program committees for several international conferences including: DAC, ICCAD, ISSS, ASP-DAC, SASIMI, and ITC-CSCC. Also, he has served as a vice chair, a secretary, and a Technical Program Committee member for a number of Japanese conferences, including: Institute of Electronics, Information and Communication Engineers of Japan (IEICE), the Information Processing Society of Japan (IPSJ), Biwako WS, and as a general chair of Karuizawa WS. He is an Associate Editor of *Transactions on IEICE on VLSI CAD*. He is also a member of IPSJ and IEICE. He received the IPSJ Convention Award in 1988, and the NEC Distinguished Contribution Award in 1993 for his logic synthesis system and in 1999 for his formal verification system.

Takumi Okamoto received the B.E. and M.E. degrees in electronic engineering from Osaka University, Osaka, Japan, in 1988 and 1990, respectively.

He joined NEC Corporation, Kawasaki, Japan, in 1990 and is currently an Assistant Manager in Computer & Communication Media Research Laboratories. From September 1995 to September 1996, he was on leave of absence from NEC as a Visiting Researcher at the Computer Science Department, University of California, Los Angeles. His research interests include VLSI physical design and combinatorial optimization algorithms.

Mr. Okamoto has served on the technical program committees of ICCAD and ASP-DAC.