IPSJ Transactions on System LSI Design Methodology Vol. 3 194–206 (Aug. 2010)

Regular Paper

A Unified Performance Estimation Method for Hardware and Software Components in Multiprocessor System-On-Chips

Arif Ullah Khan,[†1] Tsuyoshi Isshiki,[†1] Dongju Li[†1] and Hiroaki Kunieda[†1]

With the growing complexity of consumer embedded products and the improvements in process technology, multiprocessor system-on-chip (MPSoC) architectures have become widespread. These MPSoCs include not only multiple processors but also multiple dedicated hardware accelerators that can be designed from software programs, written in high-level languages like 'C', using high-level synthesis tools (HLS). Traditional techniques of HW/SW cosimulation are very slow and time consuming when used for exploring HW/SW partitioning strategies. There is a strong need for methodologies that quickly and accurately estimate the performance of such complex systems. In this paper, we present a system level performance estimation method for exploring the trade-off between hardware and software implementations in such "hybrid" MPSoC architectures. The key feature of our performance estimation is the unified timing model, in the form of a program trace graph (PTG) for both software executions on processors as well as the hardware blocks (finite state machines) synthesized by a HLS tool. The RTL code from the HLS tool is analyzed and its state transition graph is transformed into the PTG, which was originally developed for software timing annotations. These PTGs represent the workload of the computation that is driven by program execution traces in the form of 'Branch Bitstreams'. Our methodology allows highly accurate performance estimation under the existence of data dependent behavior of software and hardware components.

1. Introduction

The embedded systems for information equipment such as cellular phones, household information devices and in-vehicle information equipment are becoming increasingly sophisticated. The embedded systems of today are often developed as fully integrated systems on one single chip. As embedded systems are becoming more and more complex, in terms of speed and performance, the number of processing cores on a chip tends to increase rapidly. Next-generation embedded multimedia systems will often be built on multi-processor systems on a Chip (MPSoC) to obtain a high computing power at a relatively low energy cost. To increase the performance and achieve real time goals, hardware (HW) IP blocks are also integrated into MPSoC. MPSoC is becoming a much more prevalent design style in the quest, to achieve tight time-to-market design goals, high levels of performance and flexibility, and at the same time, low-cost and power-efficient implementations. These benefits come at the cost of higher system complexity. Traditional HW/SW co-design techniques for design space exploration and HW/SW partitioning are evolved around a uni-processor model. Now in the era of MPSoC, where an application will be divided into several SW nodes and HW nodes, new design methodologies are required to address the various design challenges [1)] in MPSoC design and architecture exploration. One of the main design challenges is the necessity for fast exploration of multiple HW and SW implementation alternatives with accurate estimations of performance, energy, and power, in order to adjust the MPSoC architecture at an early stage of the design process. In this paper we will focus on a fast unified performance estimation methodology to estimate the combined performance of multiple SW and HW bound components in an application with high accuracy. In our previous paper [2)], we have proposed the trace-driven workload simulation method for an MPSoC composed of SW components (processors) only. In our performance estimation framework, workload models in the form of program trace graphs (PTG), are automatically generated from the application source code. The program execution trace is encoded in the form of a branch bitstream that is generated by executing the instrumented source code on the host machine. Any data dependent behavior of the application can be completely reproduced with the PTG workload driven by the branch bitstream. Our methodology results in a highly accurate performance estimation. We applied the same methodology when simulating HW components. We use a novel method of extracting the timing model of the HW components generated by a commercial high-level synthesis (HLS) tool. We analyze HDL code generated by the HLS tool and back-annotate the timing information to the PTG generated from the source code. With this approach, we

†1 Tokyo Institute of Technology

can quickly estimate the performance of the target MPSoC with mixed SW and HW components. Our technique avoids the use of slow RTL simulation of HW components. The rest of the paper is organized as follows: In Section 2 we will present a brief overview of related work which will be followed by our SW/HW partitioning framework in Section 3. In Section 4 we will explain our unified performance estimation methodology for HW and SW nodes. Experimental results will be presented in Section 5, which will be followed by a conclusion and future work in Section 6.

2. Related Work

There is a need for new methodologies of fast and accurate design space exploration for MPSoC design. These new methodologies should enable an MPSoC designer to analyze different MPSoC solutions with complex embedded applications in a short period of time. These new methodologies should also be able to consider realistic inputs of the final working environment to cover the variations in data loads at runtime. Various simulation-based methods exist in the literature for estimating the performance of SW only systems using single processors. These methods include sampling-based simulation [3),4)] and hybrid simulation [5),6)]. These techniques combine native execution speed with the accuracy of ISS for some parts of the code. Another approach is to apply source-level timing annotation of the target processor for generating a timing model during software execution [7),8)]. Here, the timing information can include both static timing (obtained at compilation time) and dynamic timing (caches and branch predictions) [8)]. In the case of an MPSoC, statistical workloads have been used to model MPSoC subsystems such as bus traffic [9)] or NOCs [10)] for certain application domains such as networking [11)]. Another approach is to employ system-level simulation on the target MPSoC architecture, driven by application software with timing annotation which can evaluate the impacts of an interconnect architecture and memory subsystems in detail [12),13)]. Our cycle-accurate workload model for SW components is derived from the static timing model, whereas dynamic timing is not directly addressed in our current framework. Our simulation framework includes bus traffic modeling induced by inter-processor communications, but currently does not generate memory traffic. For early design space exploration of a system consisting of HW/SW blocks, several co-simulation techniques have been proposed, both at transaction and cycle-accurate levels, using hardware description languages (HDLs) and SystemC. Nevertheless, although these complex combined SW techniques achieve accurate estimations of the system performance, they are very slow due to the synchronization between different simulators. Moreover, higher abstraction-level simulators do attain faster simulation speeds, but at the cost of a significant loss in accuracy, hence, they are not suitable for fine-grained architectural tuning. Applying these techniques to a complex MPSoC of today and of the future, which will consist of several dozens of HW blocks and SW blocks will be either time consuming or inaccurate. Our technique enables fast design space exploration for HW/SW partitioning while eliminating the use of a HDL simulator. In Ref. 14), architectural level performance estimation for IP based systems is proposed. Their proposed architecture level estimation method is based on the construction and analysis of an Execution Dependency Graph. They assume that the timing information of the HW bound part of the application is readily available. A. Allara presented [15)], a HW/SW estimation model for TOSCA environment. They are statistically computing the timings for various SW and HW nodes; while their simulation speed is limited to few hundreds of events per CPU second. In Ref. 16), a trace driven HW/SW co-simulation technique using virtual synchronization between component simulators is introduced, achieving a simulation speed of several hundred kilo cycle/sec. The technique uses a HDL simulator to capture HW execution trace, which limits the overall simulation speed to several hundred cycles/sec. In contrast, our simulation technique does not use HDL simulator to capture HW execution, instead the execution trace is constructed by parsing the HDL code, generated by the HLS tool, which consist of an FSM. The overall simulation speed is several hundred million cycle/sec, with highly accurate system performance estimation. Another approach to enhance the simulation speed is by modeling various MPSoC components at a higher abstraction level. In Ref. 17), authors have proposed a TLM based modeling of an MPSoC for performance estimation. Their approach involves TLM modeling of various MPSoC components. Using PVT-TA model they were able to accelerate simulation speed by up to 18 times, but the accuracy of performance estimation was degraded due to communication error. Their PVT-EA model can

achieve higher accuracy at the cost of reduced simulation speed.

3. System-Level SW/HW Partitioning Methodology

3.1 Tightly-Coupled Thread Model for MPSoC Application Development

Authors have previously proposed the application development framework for an MPSoC based on the Tightly-Coupled Thread (TCT) model [18),19)]. In TCT framework a designer starts from a pure sequential C description of the application; the designer defines the system partitioning directly on the C programs. The TCT compiler performs a complete inter-procedural dependence analysis to extract all data dependencies between threads (including globals and pointer dereferences), where message-passing (communication and synchronization) instructions between various processing cores and HW blocks are inserted automatically by TCT compiler to guarantee the correct behavior of the original sequential program (the current TCT compiler assumes a fully distributed memory system without any shared-memory access). A rich tool environment of the TCT framework provides feedback to the designer which facilitates the tuning of system partitioning and the original C code to expose more parallelism. Using TCT tools, application developer can obtain an estimate of parallel execution time, sequential execution time, and communication bandwidth for each thread-to-thread connection and processor utilization. In TCT MPSoC design framework HW/SW partitioning is carried out by declaring a thread scope. A thread scope indicates a separate parallel process, which we simply refer to as threads, to be executed on a (separate) processor/HW block. Thread scope's statement syntax is given as:

THREAD (name) { statements }

Any C statement (including function calls and nested thread-scopes) can be included inside the thread-scope region as long as the thread-scope forms a Single-Entry Single-Exit (SESE) region. Thread annotations can be inserted manually or through the MAPS framework [20)]. MAPS assists the designer with rich program analysis capabilities to emit thread annotated code semi-automatically.

```
void JPEGtop(FILE *fp){
  for(i = 0; i < sizeYPadding; ){ // Loop L0
    for(ii = 0; ii < 8; ii++){ // Loop L1
      ReadOneLine(fp,i++);// RGB=>{Y0/Y1,Cb0,Cr0}
      ReadOneLine(fp,i++);// RGB=>{Y0/Y1,Cb0,Cr0}
      THREAD(Dsamp){
        DsampCbCr(i); // 4:1 Cb,Cr
      }
    }
    THREAD(BLKcore){ // call the core functions
      int nR = (i - 8 >= sizeY);
      for(j = 0; j < sizeX; j += 16){ // Loop L2
        int nC = (j + 8 >= sizeX);
        THREAD(Y0){ // process Y (upper blocks)
          BLK8x8(&Y0[j],0,&DCy,&state,0);
          BLK8x8(&Y0[j+8],0,&DCy,&state,nC);
        }
        THREAD(Y1){ // process Y (lower blocks)
          BLK8x8(&Y1[j],0,&DCy,&state,nR);
          BLK8x8(&Y1[j+8],0,&DCy,&state,nC+nR);
        }
        THREAD(C){ // process Cb/Cr-comps
          BLK8x8(&Cb[j>>1],1,&DCcb,&state,0);
          BLK8x8(&Cr[j>>1],1,&DCcr,&state,0);
        }
      }
    }
  }
}
```

Fig. 1 JPEG encoder program with thread scopes for the top function.

These tightly-coupled threads operate in a functional pipeline manner, achieving a high degree of parallelism. The TCT compiler also guarantees that the behavior of the parallelized code and the original sequential code is identical where race condition and deadlocks are automatically avoided.

Figure 1 shows an example of a C program for the top function of the JPEG encoder (modified from the original source by Ref. 21)). In function JPEG top, there are five thread scopes inserted in to the code. The base-thread handles the file input and RGB-YCbCr conversion. Thread Dsamp handles the down conversion of Cb/Cr components. Thread scope BLKcore contains three threads Y0, Y1 and C which call the core function BLK8x8 for the six component blocks. Solely inserting thread scopes may not result in speed up because of data dependency between threads. So, to improve the performance of the code we usually look for data dependencies between threads and try to remove them. This is a manual iterative process, incorporating feedback from different tools in the TCT environment, e.g., trace scheduler, call graph, program graph, and data dependency graph. We also look for parts of code inside threads which are data independent

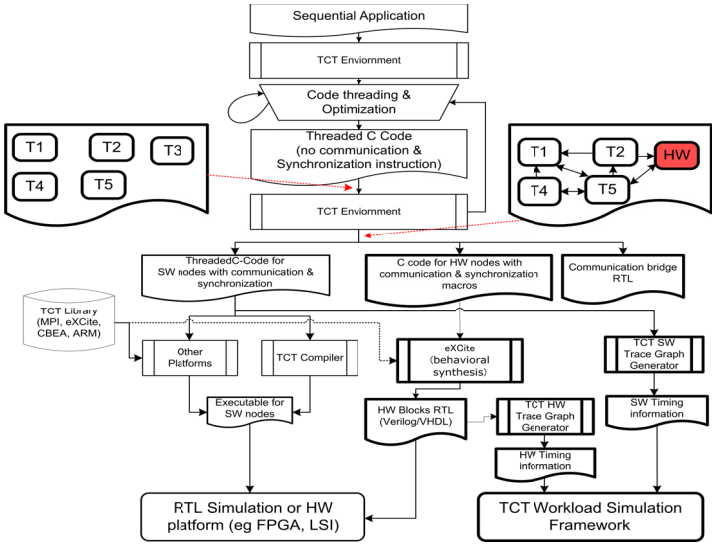

Fig. 2 TCT MPSoC design framework flow.

of each other to further extract parallelism.

3.2 TCT Communication Model and HW Generation

For some of the functionalities in application-specific MPSoCs, HW implementation may be desirable in terms of speed and area, but with the sacrifice of SW programmability. In our framework, these HW functional blocks are modeled in the same way as SW functional blocks under our TCT framework. As shown in **Fig. 2**, an application developer will start from a sequential C code and partition the code in several threads. An application developer has the freedom to assign each parallelized thread to run as a dedicated HW, or run as SW on a processor. For threads run as HW nodes, C code with communication APIs for a target HLS tool will be generated for each of the threads. The C code will be synthesized by a commercially available HLS tool "eXCite" [22] to obtain RTL code. Threads to be run as SW nodes will be compiled for TCT processor. TCT compiler also generates partitioned C source for all threads with communication and synchronization operations, which are inserted as macro calls. These communication macros can be redirected to parallel processing API calls such as MPI, and also

```
/////Output channel Definition/////
#define TCT_DT(data, size, dstPE, dstPort, srcPortID, comID)\
yx_meschan_write(__CTRL_OUT_PORT_ID__,&__COMM_ID_## comID,0);\
yx_meschan_write(__OUT_PORT_OFFSET__+ srcPortID, data, 0)

 void ThreadEntry5_0(void)
    {
     /// [DCT] startThread
     TCT_CT(6, 0, 3, 0);   // "Control Token" send to PE6
     DCTcore(__B1_5_coef,   __UB5_coef2); // Call to main DCT function
     TCT_DT(__UB5_coef2, 256, 6, 4, 5, 2); //"Data Transfer" Send coef2[64] to PE6
     /// [DCT] endThread
     TCT_LACK();
    }
void DCTcore(int * coefIn, int * coefOut){
...
    TCT_DS(__B1_5_coef, 256, 4); //"Data Synchronisation" receive coef[64]
...
}
```

Fig. 3 Sample C-code for eXCite.

to other HLS tools' communication APIs. As shown in the generated C code in **Fig. 3**, there are three communication operations defined in the TCT model:

- Control token (CT) sends an activation token to the target thread
- Data transfer (DT) sends data to the target thread
- Data synchronization (DS) checks if the data is received

Here, a DT-node does not require a matching receive operation at the receiver end, thus the data transfer occurs asynchronously with the receiver thread state. A finite-size buffer is allocated for individual data at the receiver, and the DT-node stalls while the receiver buffer for that particular data is full, where a DS-node stalls while the data buffer is empty. A CT-node is modeled in the same way as a DT-node in terms of the buffer model, where it also stalls if the receiver's control token buffer is full. Data buffers and control token buffers are consumed at thread-end node at the receiver thread.

Additionally, DT-nodes and CT-nodes are blocked if the receiver is occupied by other transactions or if the bus is occupied. These communication protocols were originally implemented in our prototype TCT-MPSoC [23], where the TCT-PE contains a dedicated communication module to implement the asynchronous (non-blocking) message passing with low setup time (4 to 6 cycles) and burst transfer of 4 bytes/cycle through a high speed interconnect realized as full crossbar connection.

The eXCite tool provides message channels with several options such as FIFOs and handshaking protocols. For interfacing eXCite message channels with TCT

communication protocols, the HDL code of a communication bridge is automatically generated by the TCT compiler along with C code for eXCite **Fig. 4**. The C code is compiled by eXCite to generate HDL code of a HW block. The bridge consists of FIFO memory and channel control logic. Bridge will first receive data or control tokens from a processor and store it in the FIFO. A read operation at the eXCite HW will read data/ control tokens from the bridge FIFOs via a blocking channel. If the read operation at the HW occurs before the bridge receives any data/control token from a processor, it will block the communication until it receives a data/control token. A write operation at the HW first sends a request token containing the target processor ID and port ID to the bridge where it is converted into a TCT request signal and sent to the target processor. When the request is acknowledged by the target processor, the bridge will raise the ACK signal of the channel and the eXCite HW will start the data transfer through a dedicated FIFO channel. **Figure 5** shows plug & play HW block for TCT MPSoC.

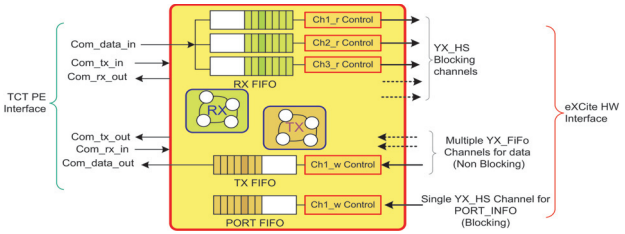

Fig. 4 TCT-eXCite communication bridge.

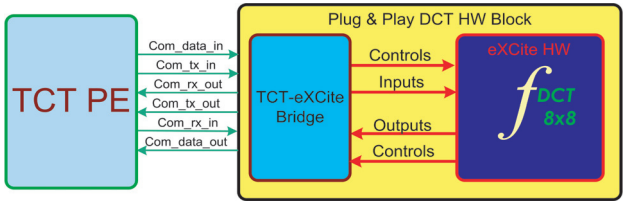

Fig. 5 Plug and Play HW block for TCT MPSoC.

4. MPSoC Performance Estimation Methodology

4.1 Trace-Driven Workload Model for SW Components

Our novel MPSoC performance estimation methodology is based on the trace-driven workload simulation framework [2]. Its key concepts are the branch bitstream, enabling efficient encoding of the program execution trace, and the program trace graph (PTG), for modeling the software execution timing. The branch bitstream is generated through source-level instrumentation for emitting branch condition bits in the execution order to a file which is then executed on a host machine.

PTG is generated by analyzing the source code and extracting timing information from the compiler's back-end code generator. An example of PTG is shown in **Fig. 6**, where a PTG-node consists of function-start, function-end, branch and call. PTG-edge corresponds to the code segment without conditional jumps, and the timing information is annotated on the PTG-edge.

Also in Fig. 6, the branch bitstream of 1011001 is shown this represents the program execution trace. This branch bitstream can be decoded back to the sequence of PTG-edges by traversing the PTG and reading the branch bitstream one bit at a time upon reaching a branch node to decide which branch path to continue traversing. The particular branch bitstream in Fig. 6 will result in a sequence (e_0 e_1 e_7 e_9 e_2 e_3 e_3 e_4 e_5 e_7 e_8 e_6). The total execution cycle count can then be simply computed as:

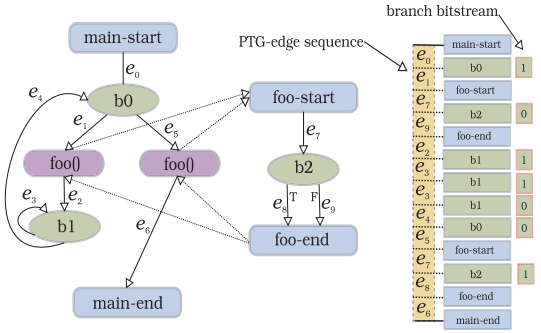

Fig. 6 Program trace graph (PTG).

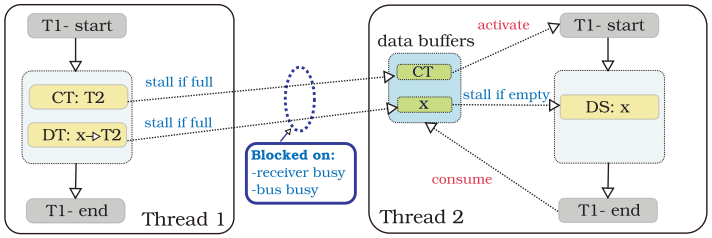

Fig. 7 Thread-PTG-sync-nodes for modeling communications.

$$T = \left.\begin{array}{l} c(e_0) + c(e_1) + c(e_7) + c(e_9) + c(e_2) + c(e_3) + \\ c(e_3) + c(e_4) + c(e_5) + c(e_7) + c(e_8) + c(e_6) \end{array}\right\} \quad (1)$$

where c(e_i) is the cycle count on PTG-edge e_i .

4.2 MPSoC Trace-Driven Workload Simulation

MPSoC applications based on TCT model consist of several threads. We construct a PTG for each Thread and call it 'Thread Program Trace Graph' (T-PTG). A T-PTG corresponds to the PTG that is enclosed by the SESE thread-scope region. As shown in **Fig. 7**, each T-PTG is terminated by thread-start and thread-end nodes instead of function-start and function-end nodes in the normal PTG. **Figure 8** shows a simplified block diagram of our Trace Workload Simulator (TWS), which consists of a global scheduler and TWS kernel. Each TCT thread includes TCT communication operations which interact with other processors through MPSoC interconnect. For modeling these system-level synchronization events, the communication operations (CT, DT and DS) appear as distinct PTG-sync-nodes in the T-PTG, where these T-PTG-sync-nodes serve as anchor points for our MPSoC workload simulator kernel to update its internal status, such as channel buffers and interconnect occupancy, and if needed, adjust its simulation clocks by inserting wait states on blocked events.

4.3 Timing Extraction of HW Components

For modeling the timing behavior of the HW components synthesized by the HLS tool, we utilize the same PTG representation generated from the original source program and extract the cycle count on each PTG-edge by analyzing the state machine description in the generated RTL code. Generated RTL contains

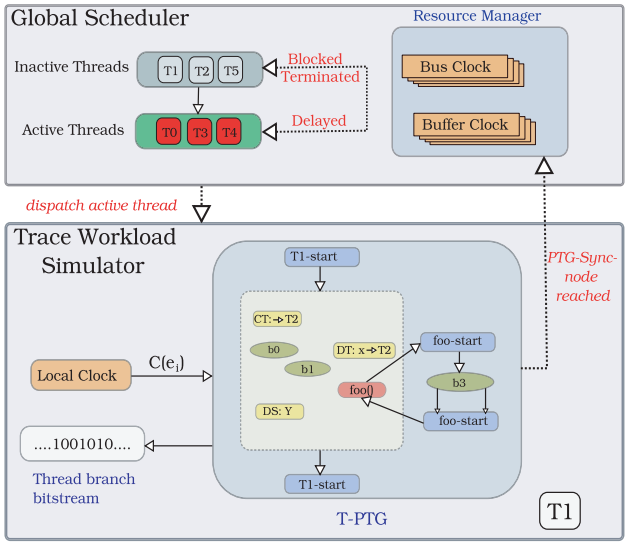

Fig. 8 MPSoC trace-driven workload simulator.

a single algorithmic state-machine and a datapath. The datapath is composed of components (e.g., adder, subtractor, shifter, etc.) taken from eXCite's database suitable for RTL synthesis and simulation. Blocking channels are used for input and output operations. We are not using pipelining option for HW generation in the HLS tool setting. All function calls except main() in a C description are inlined in the generated RTL code. In order to make the timing extraction an automated process, we rely on several assumptions and a priori knowledge of the generated RTL code (we use Verilog RTL output for this analysis). These assumptions are independent of C source code, but changes in HLS tool settings/versions may affect some of the assumptions (different names for signals and registers) which will lead to slight modifications in TCT compiler:

- Generated RTL consists of a single state machine and the order of message channel accesses in the original C code is preserved in the state machine. Also, the general control flow structure in the original C code is preserved in the RTL state transition structure.
- Signal naming conventions such as state register, channel signals, and signals

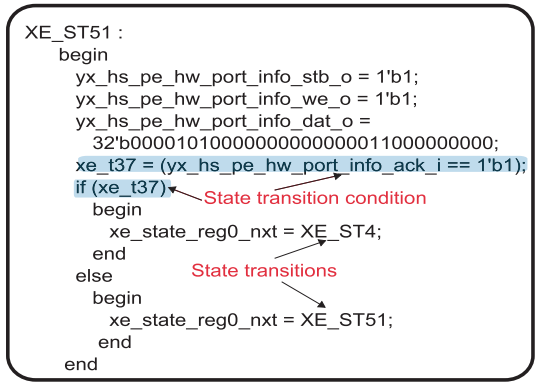

Fig. 9 Conditional state transition code.

corresponding to the original C code are known. This is essential for identifying the correspondence between the state transitions in the RTL and the control flow in the original C code.

- State transition structures arising from message channel accesses are known for each channel type. These channel access state transitions correspond to TCT communication operations.

This timing extraction is conducted using the following steps:

(1) `State transition graph extraction:` RTL code is first parsed and the state transition information including the state transition condition expressions are extracted to construct the state transition graph (STG). **Figure 9** shows a section of generated RTL code with condition state transition from state XE_ST51 to XE_ST4. The code corresponds to TCT communication operations and in this case, the state transition condition is determined by the acknowledge signal (yx_hs_pe_hw_port_info_ack_i) from the bridge.

(2) `STG reduction:` Unconditional state transition edges (STG-edges) are collapsed and the number of consecutive unconditional STG-edges is annotated to the conditional STG-edge that connects to the first unconditional STG-edge. This reduced STG, in essence, has the same structure as PTG with additional STG edges representing message channel accesses.

(3) `Communication node extraction:` A group of STG-edges having conditional expressions on channel signals are extracted and matched against known state transition structures for TCT communication operations (CT, DT, DS). Figure 9 shows a section of generated RTL for a CT operation. Such STG-edge group is replaced by the corresponding communication state node, further reducing the STG.

(4) `Timing annotation on the PTG from reduced STG:` The reduced STG has a very similar structure to the PTG constructed from the original C code, with one significant difference, that is, function calls are inlined in the state machine (in the case of eXCite tool). Distinctive PTG nodes (thread-start, thread-end, CT, DT and DS nodes) can be directly matched against the communication nodes in the reduced STG by examining their state transition condition channel signals. As discussed previously, the sequence of message channel accesses is assumed to be preserved, and under this assumption, the matching of PTG and reduced STG becomes a fairly straightforward task of traversing both graphs starting from thread-start node while circumventing the differences in call structures. After identifying the matching edges, the cycle count on the STG-edge, i.e. the number of unconditional state transitions, are back annotated to the PTG-edge.

Figure 10 (a) shows the STG of the RTL generated from the C code in Fig. 3 where the unconditional STG-edges are collapsed and cycle counts are annotated on the conditional STG-edges. State transition conditions with hsXXack indicate the message channel handshaking (XX indicates the channel ID) which corresponds to one of the TCT communication operations. The following eXCite channels are used for various TCT communications:

- hs100: read channel for receiving the control token for thread activation (state S51 in the Fig. 10 (a))
- hs14: read channel for receiving array data coef [64] (state S48)
- hs1000: write channel for CT, DT request or signaling thread termination. Their distinction can be determined by the constant values written on this channel. Here, state S50 corresponds to CT operation (sending control tokens to other processors), S47 corresponds to a DT request on the output data coef2 [64], and S43 corresponds to thread termination. For DT operation, after the DT request is acknowledged, additional write FIFO channel is used

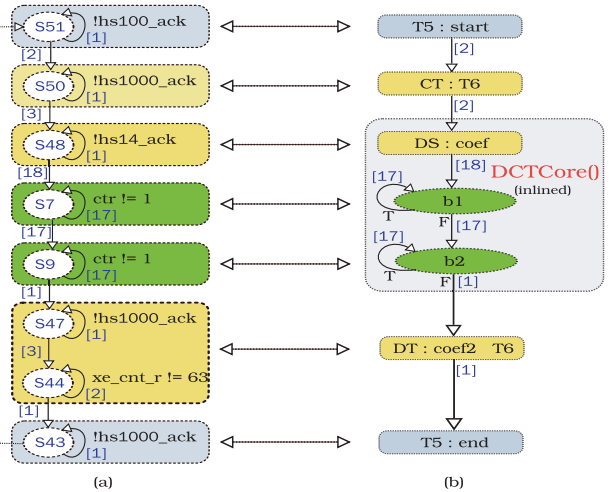

Fig. 10 (a) Reduced state transition graph (b) program trace graph.

with a counter for controlling the burst transfer of array data (state S44).

After identifying the channel handshaking state transitions, and matching with the TCT communication operations, we perform graph matching against the corresponding PTG in Fig. 10 (b), generated directly from the original C code and annotate the timing information (cycle counts on STG-edges) to the PTG-edges.

4.4 Discussion

On nearly all of our test cases so far, our HW timing extraction methods described in the last section have been successful in completing the graph matching between the reduced STG from the RTL analysis and PTG from the original C code. As described earlier, we rely on the assumption that the message channel access order in the C code is preserved in the RTL state machine. This assumption, together with the knowledge of the naming conventions for eXCite channel signals, allows us to precisely identify the TCT communication operations on the HDL code. We believe that this assumption is valid for most HLS tools since changing the channel access order may easily destroy the higher level communication protocol between the SW and HW, therefore HLS tools must be conservative in implementing the channel access state transitions. On the other hand, the core control flow structure of the application, especially the conditionals in the C code, also needs to be preserved in the RTL as well. Although this was the case for the tested eXCite HLS tool under the default synthesis settings, we may expect to see some aggressive optimization techniques in HLS which alter the control flow structures in the original C code which will make the graph matching problem, between the STG and PTG, non-trivial. This issue shall be addressed in our future works. Although the current implementation of our TWS framework is targeted at the TCT model and its MPSoC architecture model, we believe that the concept of PTG-sync-nodes that bridges the PTG semantics and the TWS kernel can eventually be extended to other MPSoC programming models and MPSoC architecture models. Synchronization primitives used in common parallel programming models (MPI, OpenMP) and RTOS-based multitask programming can be modeled as PTG-sync-nodes, where the TWS kernel shall be extended to handle the behavior of these additional PTG-sync-node types. Complex MPSoC architectures with memory subsystems and NOCs would require more substantial extensions on the TWS kernel for generating accurate data traffic induced by memory accesses and through the NOC infrastructure.

5. Experimental Results

To confirm the effectiveness of our proposed methodology we have conducted a series of experiments on the single-processor models, MPSoC as well as Hybrid MPSoC which include SW and HW components and have compared them against the instruction-set simulator (ISS) for TCT-PE and RTL simulation for each type of HW/SW combinations.

5.1 Cycle Estimation of SW Components

Table 1 shows the set of benchmarks used to evaluate the cycle estimation performance of the single-processor model. Here, "# inst" is the number of TCT-PE instructions in each program, "# cycles" is the total number of execution cycles, "ISS (sec)" is the ISS simulation time, "ISS (cycles/sec)" is the ISS simulation speed, and "native (sec)" is the native execution time (all applications were compiled with Microsoft Visual C++ .NET with -O2 optimization). **Table 2** shows our trace-driven workload simulator (TWS) performance without PTG reduc-

Table 1 Benchmark programs for single-processor cycle estimation.

	#inst	#Cycles	ISS (sec)	ISS (cycles/sec)	native (sec)
prime500k	100	629,940,586	13.328	47.26M	0.083
string	273	900,232,697	31.266	28.79M	0.172
alphabet	113	716,650,704	25.328	28.29M	0.193
JPEG	1,673	433,010,644	16.829	25.73M	0.076

Table 2 Trace-driven workload simulator (TWS) performance (without PTG reduction).

	#PTG-edges	#branch bits	TWS (sec)	TWS (cycles/sec)	TWS speedup
prime500k	29	53.99M	0.218	2,889M	61.14
string	52	136.18M	0.515	1,748M	60.71
alphabet	21	101.04M	0.375	1,911M	67.5
JPEG	336	27.48M	0.141	3,071M	119.35

tion. Here, "# PTG-edges" is the total number of PTG-edges, "# branch bits" is the length of the branch bitstreams, "TWS (sec)" is the TWS simulation time, "TWS (cycles/sec)" is the effective TWS simulation speed, and "TWS speedup" is the speedup over ISS. Even without the PTG reduction, we can observe the enormous speed advantage of our TWS over ISS, where the effective TWS simulation speed ranges from 1.7 to 3.0 billion cycles per second, while achieving absolute cycle accuracy, that is, the TWS cycles, counts match exactly with the ISS cycle counts.

Figure 11 shows the processing times inside the TWS workflow that are normalized against the native execution time in Table 1. Here, "BB-gen" is the native execution time of the applications instrumented with branch bitstream generation, and "BB-gen+div" is the execution time with branch bitstream generation and integer division latency profiling. We can observe that the instrumentation overhead for the branch bitstream generation is quite small, ranging from 1.40 to 2.25, and even with the division latency profiling, the overhead increases only by up to 2.75. "BB-scan" is the processing time of enumerating the PTG-edge occurrences from the generated branch bitstreams that is performed prior to the PTG reduction. Each of these applications in this single-processor case reduces to a single PTG-edge whose edge cycle match exactly with the ISS

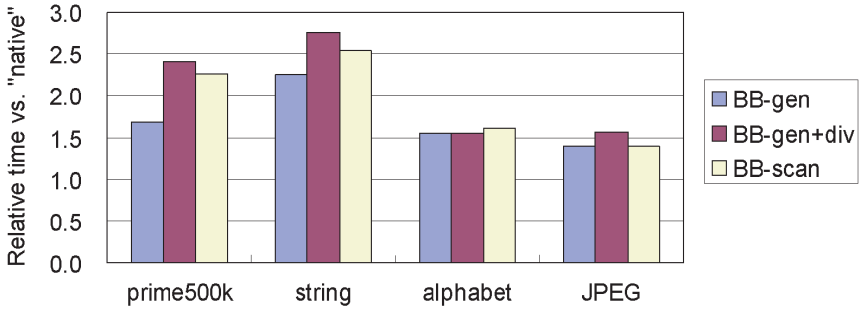

Fig. 11 Relative processing times against native execution.

Table 3 Cycle count for hw blocks.

Thread	#Cycles HDL Sim.	#Cycles Estimated	Estimation Error%*1
DCT	424	412	2.83
RGB	1877	1813	3.40

cycle count.

5.2 Cycle Estimation of HW Component

To validate our method for HW cycle estimation we conducted a series of experiments using a JPEG encoder. This application was chosen because it is a real-world multimedia processing problem, its complexity is not-so-high, and it possesses enough features that can be used to verify our design methodology. We used a reference program made available by the Independent JPEG Group [8]. We partitioned the sequential code into 8 parallel threads. We selected DCT and RGB threads to run as HW nodes. Estimated clock cycles for a single iteration of 8x8 DCT by our TWS simulator were 412, while from HDL simulation 424 cycles were recorded. Similarly we got 1813 cycles for conversion of 384 RGB pixels to YCbCr color space from our TWS simulator, while from the HDL simulator 1877 cycles were recorded, as shown in **Table 3**.

5.3 Hybrid MPSoC Cycle Estimation

To confirm the effectiveness of our methodology for overall system performance estimation we setup a HW platform for RTL simulation. Our test platform consists of 8 TCT processors [23] connected via a full cross bar interconnects having

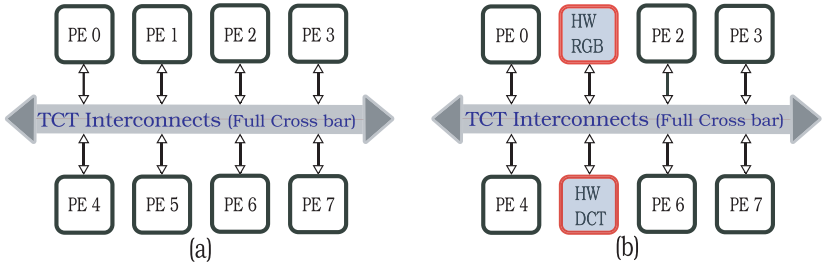

Fig. 12 Testing platforms.

Table 4 Simulation speed and accuracy.

#Threads SW	#Threads HW	#Cycles HDL	#Cycles TWS	Sim. Time HDL (Sec)	Sim. Time TWS (Sec)	Estimation Error%⋆1
1	0	4503635	4486414	156	0.0	0.49
8	0	1221215	1191293	154	0.016	2.45
7	1(DCT)	1219175	1186627	291	0.0	2.67
7	1(RGB)	1050984	1011625	280	0.0	3.75
6	2(DCT+RGB)	1049431	1002200	234	0.0	4.50

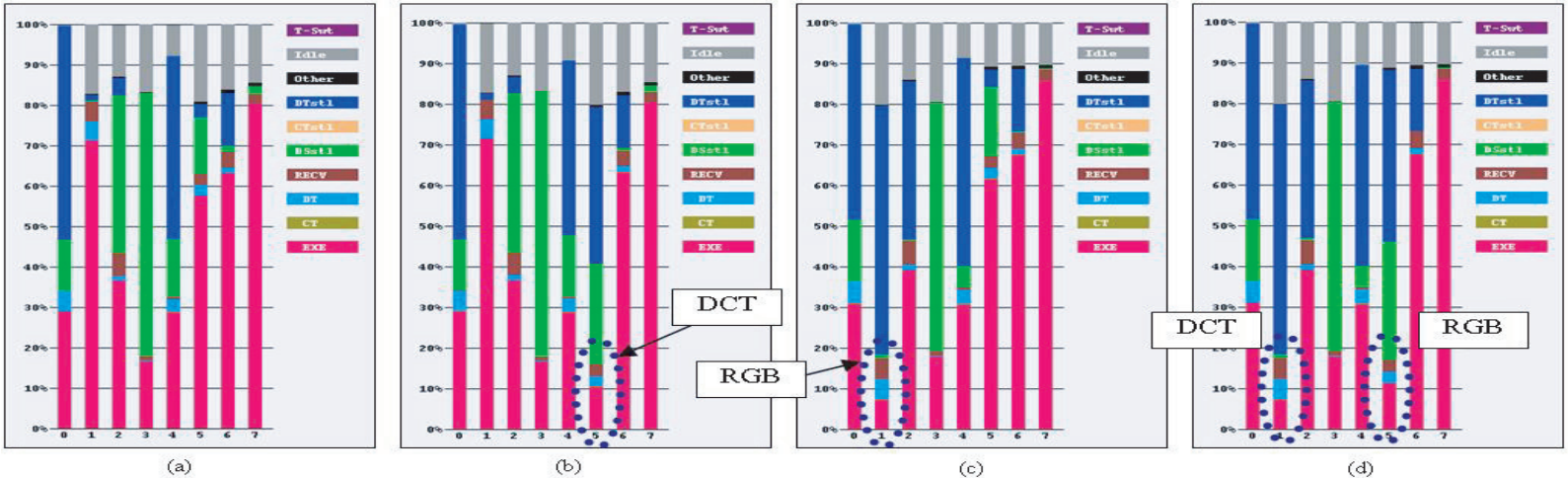

Fig. 13 Execution profiles of the 8 threads (a) 8 SW threads (b) 7 SW + DCT-HW (c) 7 SW + RGB-HW (d) 6 SW + DCT-HW + RGB-HW (THREAD #1 = RGB-thread, THREAD #5 = DCT-thread).

decentralized autonomous arbitration as shown in **Fig. 12**. First we run the partitioned application using 8 TCT processors Fig. 12 (a). In the second step some processors were replaced by HW blocks, which were generated by the method explained in Section 3, as shown in Fig. 12 (b). We conducted RTL simulations for various HW/SW combinations, from running software on a single PE to using multiple HW blocks. Our RTL simulation results are shown in **Table 4**. Later, we used our trace workload simulator to simulate similar HW/SW co-design strategies. The data from our workload simulator is also quoted in Table 1.

Although the maximum difference between the total cycle count estimated with our TWS and the cycle count from the RTL simulator is around 4.5%, the difference between estimated overall performance from an RTL simulator and our workload simulator is quite small for different partitioning schemes. One of the

⋆1 Estimation Error = (HDL Cycles-Estm. Cycle)/HDL Cycle * 100

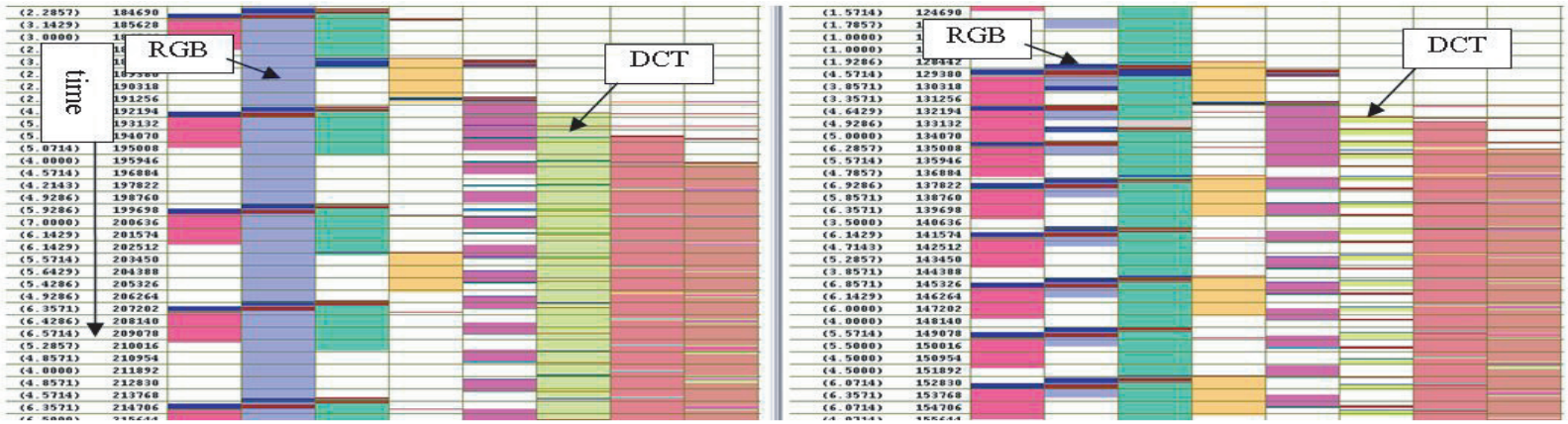

Fig. 14 Execution schedule extracted from the trace-driven workload simulator (left: 8 SW threads, right: RGB-HW + DCT-HW).

reasons for the difference between RTL and the estimated cycle count is due to not considering the communication delay, caused by buffering inside the bridge, in our workload simulator.

With our simulation methodology we achieve accuracy as well as remarkable simulation speeds as shown in Table 4. **Figure 13** shows the execution profiles of the 8 threads under different configurations. Here, the bottom red portion of the bar graphs corresponds to the computation time, where other colors correspond mainly to the idle times incurred by thread synchronization. The speedup of the individual threads can be clearly seen, however these speedups of the individual threads through HW implementation do not quite contribute to the overall performance. This can be verified in **Fig. 14**, which shows the execution schedule extracted from the trace-driven workload simulator, where the left view shows the 8 SW thread implementation and the right view shows the DCT-HW+RGB-HW implementation. When these two threads are sped up by HW implementations, other threads become the overall bottleneck. Our trace-driven workload simulation framework provides analysis capabilities that offer significant assistance in aiding the designer when visualizing the important design decisions.

6. Conclusions and Future Work

In this paper we proposed a novel performance estimation methodology based on a unified timing model of HW and SW components using program trace graphs. Our experimental result shows that the results are very close to the simulation of RTL or ISS while achieving a remarkable speed advantage over RTL and ISS simulation. Our technique enables us to quickly estimate the performance of various types of HW/SW partitioning. Our current research was focused on estimating timing performance. In future we will focus on the fast estimation of power consumption and area usage for different combinations of HW and SW. We will also enhance our tool set to give a designer a more detailed report of system performance in terms of speed, power and area for different HW and SW combinations. In future we will also address the limitations, discussed in Section 4.4, of our current implementation.

References

1) Martin, G.: Overview of the MPSoC Design Challenge, *Proc. DAC' 06*, pp.274–279 (2006).
2) Isshiki, T., et al.: Trace-Driven Workload Simulation Method for Multiprocessor

System-On-Chips, *Proc. DAC '09*, pp.232–237 (2009).
3) Wunderlich, R., et al.: SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling, *Proc. 30th ISCA*, pp.84–97 (2003).
4) Sherwood, T., et al.: Automatically Characterizing Large Scale Program Behavior, *Proc. ASPLOS-X*, pp.45–57 (2002). (distributed to authors).
5) Szwed, P.K., et al.: SimSnap: Fast-Forwarding via Native Execution and Application-Level Checkpointing, *Proc. INTERACT-8*, pp.65–74 (2004).
6) Kraemer, S., et al.: Hysim: A Fast Simulation Framework for Embedded Software Development, *Proc. CODES-ISSS*, pp.75–80 (2007).
7) Lazarescu, M.T., et al.: Compilation-based Software Performance Estimation for System Level Design, *Proc. HLDVT00*, pp.167–172 (2000).
8) Schnerr, J., et al.: High-Performance Timing Simulation of Embedded Software, *Proc. DAC '08*, pp.290–295 (2008).
9) Giorgi, R., Prete, C.A., Prina, G. and Ricciardi, L.: A Workload Generation Environment for Trace-Driven Simulation of Shared-Bus Multiprocessors, *Proc. 30th HICSS*, pp.266–275 (1997).
10) Madsen, J., et al.: Network-on-chip Modeling for System-level Multiprocessor Simulation, *Proc. 24th RTSS03*, pp.82–92 (2003).
11) Thiele, L., et al.: A Framework for Evaluating Design Tradeoffs in Packet Processing Architectures, *Proc. DAC '02*, pp.880–885 (2002).
12) Kempf, T., et al.: A SW Performance Estimation Framework for Early System-level-design Using Fine-grained Instrumentation. *Proc. DATE 06*, pp.468–473 (2006).
13) Meyerowitz, T., et al.: Source-level Timing Annotation and Simulation for a Heterogeneous Multiprocessor, *Proc. DATE 08*, pp.276–279 (2008).
14) Ueda, K., et al.: Architecture-level Performance Estimation for IP-based Embedded Systems, *Proc. DATE '04*, pp.1002–1007 (2004).
15) Allara, A., et al.: System-Level Performance Estimation Strategy for SW and HW, *Proc. ICCD*, pp.48–53 (1998).
16) Kim, D.: Trace-Driven HW/SW Co-simulation Using Virtual Synchronization Technique, *Proc. DAC' 06*, pp.345–348, (2005).
17) Atitallah, R.B., et al.: An MPSoC Performance Estimation Framework Using Transaction Level Modeling, *Proc. RTCSA '07*, pp.525–533 (2007).
18) Urfianto, M.Z., et al.: Decomposition of Task-level Concurrency on C Programs applied to the Design of Multiprocessor SoC, *IEICE Trans.Fundamentals*, Vol.E91-A, No.7, pp.1748–1756 (2008).
19) Isshiki, T., et al.: Tightly Coupled Thread Model: A New Design Framework for Multiprocessor System on Chips, *DA Symposium*, Shizuoka, Japan (12–13 July 2006).
20) Ceng, J., et al.: MAPS: An Integrated Framework for MPSoC Application Parallelization, *Proc. DAC '08*, pp.754–759 (2008).
21) Independent JPEG group, www.ijg.org.
22) eXCite virtual platform reference manual, Y Exploration Inc. http://www.yxi.com
23) Urfianto, M.Z., et al.: A Multiprocessor SoC Architecture with Efficient Communication Infrastructure and Advanced Compiler Support for Easy Application Development, *IEICE Trans. Fundamentals*, Vol.E91-A, No.4, pp.1185–1196, (2008).
24) Jerraya, A., et al.: *Multiprocessor system on Chip*, Elsevier Morgan Kaufmann, San Francisco, California (2005).

(Received December 1, 2009)
(Revised February 26, 2010)
(Accepted April 14, 2010)
(Released August 16, 2010)

(Recommended by Associate Editor: *Hiroyuki Tomiyama*)

Arif Ullah Khan received his B.Sc. degree in Electrical Engineering from NWFP University of Engineering and Technology Pakistan in 2001 and M.Sc. degree in Information and Communication Engineering from University of Karlsruhe Germany in 2004. He is currently working as a researcher at Department of Communication and Integrated Systems at Tokyo Institute of Technology. His interests include MPSoC design framework, NOC and HW/SW integration in MPSoC. Mr. Arif is a member of IEEE and IPSJ.

Tsuyoshi Isshiki has received his B.E. and M.E. degrees in Electrical and Electronics Engineering from Tokyo Institute of Technology in 1990 and 1992, respectively. He received his Ph.D. degree in Computer Engineering from University of California at Santa Cruz in 1996. He is currently an Associate Professor at Department of Communications and Integrated Systems in Tokyo Institute of Technology. His research interests include MPSoC programming framework, high-level design methodology for configurable systems, bit-serial synthesis, FPGA architecture, image processing, fingerprint authentication algorithms, computer graphics, and speech synthesis. Dr. Isshiki is a member of IEEE CAS, IPSJ and IEICE.

Dongju Li received her B.S. degree from LiaoNing University and M.E. degree from Harbin Institute of Technology, China, in 1984 and 1987, respectively. She worked as an IC design engineer in VLSI Design Laboratory of Northeast Micro-electronics Institute, Electronic Industry Burean, China, from 1987–1993. She is currently a Research Associate at Department of Communications and Integrated Systems in Tokyo Institute of Technology. Her current research interests are in embedded fingerprint authentication algorithms, VLSI architecture and design methodology, system on chip design for multimedia processing including video CODEC. Dr. Li is a member of IEEE CAS and IEICE.

Hiroaki Kunieda was born in Yokohama in 1951. He received his B.E., M.E. and Dr. Eng. degrees from Tokyo Institute of Technology in 1973, 1975 and 1978, respectively. He was a Research Associate in 1978 and an Associate Professor in 1985, at Tokyo Institute of Technology. He is currently a Professor at Department of Communications and Integrated Systems in Tokyo Institute of Technology. He has been engaged in researches on distributed circuits, switched capacitor circuits, IC circuit simulation, VLSI CAD, VLSI signal processing and VLSI design. His current research focuses on fingerprint authentication algorithms, VLSI multimedia processing including video CODEC, design for system on chip, VLSI signal processing, VLSI architecture including reconfigurable architecture, and VLSI CAD. Dr. Kunieda is a member of IEEE CAS, SP society, IPSJ and IEICE.