Hardware-software Codesign of Multimedia Embedded Systems: the PeaCE Approach

Soonhoi Ha, Choonseung Lee, Youngmin Yi, Seongnam Kwon, and Young-Pyo Joo
The CAP Laboratory, Seoul National University
{sha,cslee,ymyi,ksn,joo47}@iris.snu.ac.kr

Abstract

Hardware/software codesign involves various design problems including system specification, design space exploration, hardware/software co-verification, and system synthesis. A codesign environment is a software tool that facilitates capabilities to solve these design problems. This paper presents the PeaCE codesign environment mainly targeting for multimedia applications with real-time constraints. PeaCE specifies the system behavior with a heterogeneous composition of three models of computation. The PeaCE environment provides seamless co-design flow from functional simulation to system synthesis, utilizing the features of the formal models maximally during the whole design process. Preliminary experiments with real examples prove the viability of the proposed technique.

1. Introduction

As improvement of implementation technology is not slowed down, design of multimedia embedded systems becomes more challenging due to the increasing system complexity as well as relentless time-to-market pressure. Conventional practice of designing a multimedia embedded system performs algorithm design, hardware design, and software integration sequentially. Past design experiences and some performance profiling techniques are resorted to determine a hardware architecture. After a hardware prototype is made, developed software codes are downloaded into the programmable processors and verified. Since the expected growth rate of design productivity in this conventional design method is far below that of system complexity, hardware/software (HW/SW) co-design has emerged as a new design methodology in the past ten years.

As a systematic system-level design methodology, HW/SW co-design methodology no longer separates hardware design and software design. To determine the hardware architecture, we explore a wide range of feasible architectures and evaluate each one by estimating the expected performance after considering all subsequent design decisions such as hardware/software partitioning and software implementation. A systematic design space exploration needs separate specification of functions and architectures. An explicit mapping step maps parts of functional specification to architecture building blocks. After an optimal architecture is found and mapping is decided, more accurate performance verification is needed before final hardware implementation. Thus HW/SW co-design includes various design problems including system specification, design space exploration, performance estimation, HW/SW co-verification, and system synthesis. A co-design environment is a software tool that facilitates capabilities to solve those design problems.

In summary, we identify the following three as the key subjects in the HW/SW codesign methodology: system level specification, systematic design space exploration, HW/SW co-simulation/co-verification. Among these, most commercial codesign tools are concerned about HW/SW co-simulation/co-verification since it is the most urgent need to develop and verify software on a virtual prototyping environment before a hardware prototype is made. In particular, TLM(Transaction Level Model)-based co-simulation tools have been successfully used to increase the design productivity [12]. On the other hand, there are few tool, if any, available for the other subjects. In the current practice of codesign methodology, architecture exploration is usually made manually.

In this paper, we introduce a full-fledged codesign environment, called PeaCE[1], that provides seamless co-design flow from functional simulation to system synthesis, mainly targeting for multimedia applications with real-time constraints. Moreover, PeaCE is developed as a re-configurable framework to which a 3rd party design tool can be easily integrated. In the current implementation Seamless CVE of Mentor Graphics is integrated for HW/SW co-verification.

In the next section, we overview the codesign flow of PeaCE. And three key subjects of codesign methodology in PeaCE will be discussed in sections 3, 4, and 5 respectively. Section 6 shows some preliminary results with real examples and section 7 concludes the paper.

2. Proposed HW/SW codesign flow

Figure 1 illustrates the outline of HW/SW codesign procedure in the PeaCE environment.

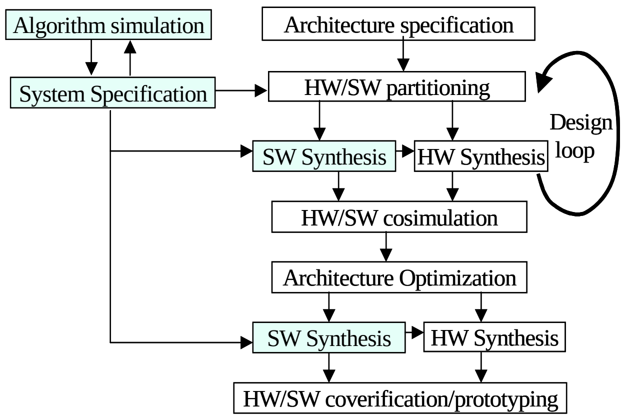


Figure 1. HW/SW codesign procedure

HW/SW co-design process starts with system specification. While a sequential "C" code description has been widely used for functional simulation in conventional design procedure, it is not adequate for initial system specification in the system level design. Since a system is implemented as a collection of concurrent components such as programmable processors, ASICs, and discrete hardware components, models of computations that express the concurrency naturally are preferred for initial specification.

In the proposed codesign methodology, the design flow begins with specification of the system behavior with formal models: a dataflow model for computation and an FSM model for control module of the system. At the top level, a task model is used to model the interaction between tasks. Then, algorithm simulation of system behavior is performed after generating a C code from the specification: the generated C code is compiled and run on the host machine.

On the other hand, we specify the candidate architectures to be explored. We may select a predefined hardware platform or create a new platform using processor cores and hardware IPs that are available in the design library.

The next step is to map the system behavior to an optimal hardware architecture. In this step, we perform HW/SW partitioning and component selection at the same time. After partitioning decision is made, PeaCE generates the partitioned codes for each processing element and co-simulates the system to generate the memory traces from the processing elements. Since the communication architecture is not determined yet, it should be noted that the co-simulation does not produce accurate performance estimation results but memory traces only. Using the memory traces and schedule information, we explore the communication architectures: as of now, bus architectures are only explored.

After the communication architecture is determined, we verify the final architecture before synthesis. This step performs time-accurate HW/SW co-simulation for co-verification. We may use Seamless CVE in this step or our co-simulation tool used. Finally, we generate the codes for the processing elements in a prototyping board. For a processor core, we generate a C code and a VHDL code for FPGA hardware implementation.

3 System behavior specification

As a test vehicle for system level specification, Figure 2 shows a multi-mode multimedia terminal (MMMT) system. An embedded system is called multi-mode when it supports multiple applications by dynamically reconfiguring the system functionality.

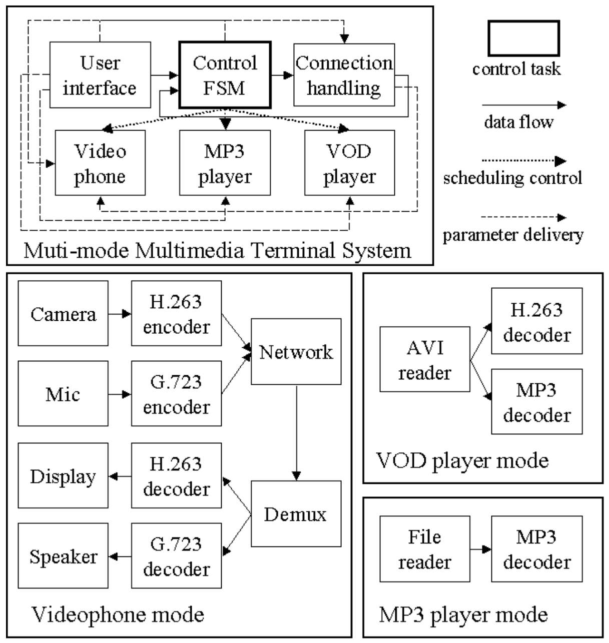


Figure 2. Multimode Multitasking Multimedia Terminal (MMMT) example

At the top level, three hierarchical nodes are defined to specify each mode of operation: a videophone, a VOD (Divx) player, and an MP3 player. Three other tasks are also specified at the top level for user interface, connection handling, and task execution control. And, each mode of operation consists of multiple tasks as shown in the figure. An example operation scenario is as follows. (1) A user determines the mode of operation with arguments, for instance the videophone mode with a host address name. The user interface task reads and delivers this information to the control task and to the connection handling task; (2) The control task now activates the mode by activating all the tasks that compose the selected mode after deactivating the previous mode; (3) The connection handling task takes care of network management such as connection establishment and message exchange; (4) The control task listens to the status signal from the current mode of operation and stops all the component tasks when termination or exception signal is received. We distinguish data, control, and parameter flows between tasks in Figure 2 while only data flow arcs are visible in the real implementation.

Behavioral level specification of such an MMMT system is very challenging with the following key requirements;

(1) Tasks of MMMT system have diverse execution semantics: data-driven, event-driven, and time-driven. The connection handling task is waked up in the event-driven fashion while the decoder tasks are usually time-driven for the constant rate of output production.

(2) There are various kinds of interactions between tasks with different synchronization requirements. Scheduling control information from the control task to a mode is asynchronous while data flow between tasks should be delivered in a synchronous fashion. Parameter delivery is also asynchronous.

(3) A single task may need to be partitioned into multiple processing components to meet the timing constraint. For example, the motion estimation block of H.263 encoder task should be mapped to a hardware block. Therefore functional decomposition of the inside of a task is desirable for design space exploration.

PeaCE uses dataflow model and FSM model to specify the internal behavior of a signal processing task and a control task respectively. There are several benefits of using formal models for specification as observed by many researchers[2]: (1) Static analysis of the specification allows one to verify the correctness of the functional specification. (2) The specification model can be refined to an implementation to ease the system validation by the principle of "correct-by-construction". (3) A formal specification model is not biased to any specific implementation method, so allows one to explore the wider design space. (4) It represents the system behavior unambiguously so that collaboration and design maintenance can be accomplished easily.

Nonetheless using formal models of computation has not gain wide acceptance mainly because of limited expression power and inefficient synthesis results. We overcome the problems of limited expression capability and of inefficiency by extending the existent dataflow and FSM models.

3.1 Extended dataflow model

In a hierarchical dataflow program graph, a node, or a block, represents a function that transforms input data streams into output streams. An arc represents a channel that carries streams of data samples from the source node to the destination node. The number of samples produced (or consumed) per node firing is called the output (or the input) sample rate of the node. In case the number of samples consumed or produced on each arc is statically determined and can be any integer, the graph is called a synchronous dataflow graph (SDF)[3] that is widely adopted in numerous DSP design environments.

But, the SDF model has a couple of serious limitations to represent multimedia applications: First, the SDF model cannot express the data dependent node executions such as "if-then-else" and "for" constructs. Second, the efficiency of the synthesized code is noticeably worse than the hand-optimized code in terms of memory requirements. Therefore, we make two extensions to the SDF model to overcome these problems, which are fractional rate dataflow (FRDF) for efficient code generation minimizing buffer size and synchronous piggybacked dataflow (SPDF) to express the global states and data dependent execution. For detailed discussion on the extended models, refer to [4] for FRDF and [5] for SPDF.

3.2 Extended FSM model

The pure FSM representation suffers from *state explosion problem* to represent a system with concurrent modules or memory. To avoid this problem, we devise an extended FSM model, called flexible FSM (fFSM), to express the state transition behavior of a control task [6]. The proposed fFSM model supports concurrency, hierarchy, and internal event mechanism as Harel's Statechart does. Unlike Statechart, however, we forbid inter-hierarchy transition to make it compositional.

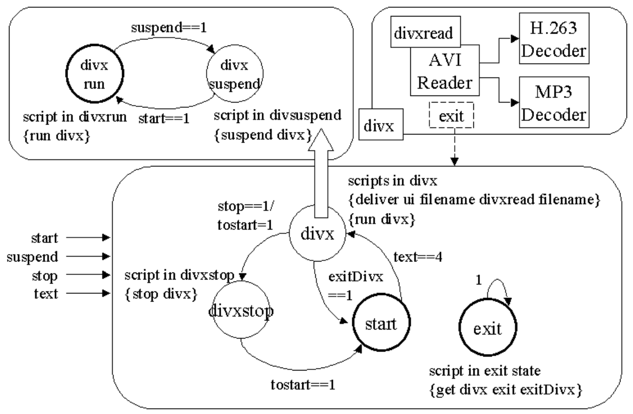


Figure 3. Control task specification based on the fFSM model in Divx mode

Figure 3 shows an example FSM to control the VOD(divx) mode, which consists of two concurrent FSMs and one hierarchical FSM and gets four inputs from a user interface task. Figure 3 also describes the task model of the divx mode Similar to the statechart, a state may have action scripts to define the control interactions between the FSM and the computation tasks. The action scripts are executed only when the state transition occurs. A bolded circle represents the default state of the FSM. More formal treatment of fFSM model can be found in [6].

3.3 Task-level specification

Tasks in the MMMT system have diverse activation conditions and port semantics that should be clearly specified in the task-level specification model at the top-level. For this purpose, a novel task-level specification model, shortly task-model, is devised. In the proposed task model, synchronization and communication between tasks are performed at only ports as message boundary. Each task accesses shared resources explicitly through ports and internal states are not directly accessible outside the task. While this task model is somewhat restricted compared with the general task model assumed in an operating system, it is free from priority inversion problem and race condition. So task scheduling and synchronization between tasks are greatly simplified. Note that data flow and FSM models satisfy these restrictions.

We support three types of tasks depending on the triggering condition: periodic tasks triggered by time, sporadic tasks triggered by external IO, and function tasks triggered by data. And we also define various port semantics by combining port type, data size, and data rate.

Remind that the internal behavior of a task is represented as a dataflow model or FSM model. On the other hand, the task itself has diverse execution semantics. So we provide the mechanism to connect the internal SDF and FSM models to the outer task-model. A task wrapper is created at the boundary of hierarchical block(or a task block) that translates the outer execution semantics to internal execution semantics.

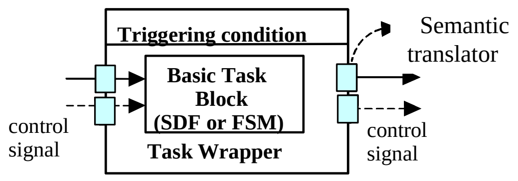
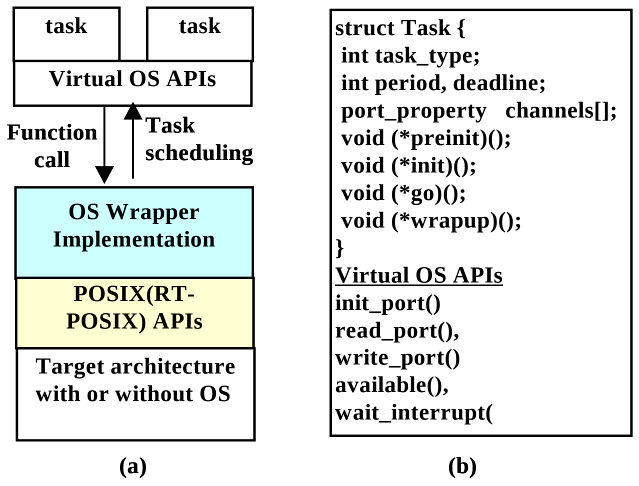


Figure 4. Task wrapper

It also contains a semantic translator for each port of the internal model. For example, if a dataflow task is defined as a periodic task with a static-size static-rate buffer port, arrival of input data is ignored while the arrived data is stored in the port buffer. Instead the current values stored in the input port buffers are delivered to the inside at periodic wake-up.

Depending on the design step, simulation or implementation, we synthesize different versions of software code from the same task-model while the inside of a dataflow or control task is unchanged. Such reconfiguration is achieved by a layered software structure as shown in Figure 5. The task structure as shown in Figure 5(b) is automatically generated from the task wrapper for each task. It is a general task code structure that can be tailored to different software implementations. Task types and port properties from the task-model are defined in the structure. A main task code is divided into four methods to support dynamic reconfiguration; preinit() is called once when the system starts, init() initializes internal variables when a task is initialized, go() contains the main task body and wrapup() is called when the system ends. In the generated code of each task, virtual OS APIs are used to access ports. The virtual OS APIs are implemented by the OS wrapper according to the port semantics and task scheduling policy.



```
struct Task {
 int task_type;
 int period, deadline;
 port_property   channels[];
 void (*preinit)();
 void (*init)();
 void (*go)();
 void (*wrapup)();
}
Virtual OS APIs
init_port()
read_port(),
write_port()
available(),
wait_interrupt(
```

Figure 5. (a) Layered software structure, (b) task structure and virtual OS APIs

While the OS wrapper can be implemented with diverse methods, the current implementation uses pthread library, especially with RT-POSIX APIs for accurate timing services. Figure 6 shows a simplified implementation of the OS wrapper. Each task is created as a separated thread by the OS wrapper. The scheduling control task receives control signals from the FSM task and determines the status of the tasks. Scheduling action is performed by the *scheduler* task that emulates the scheduling policy in the target operating system. Note that division of the scheduling control task and the scheduler task is not obligatory in the OS wrapper implementation.

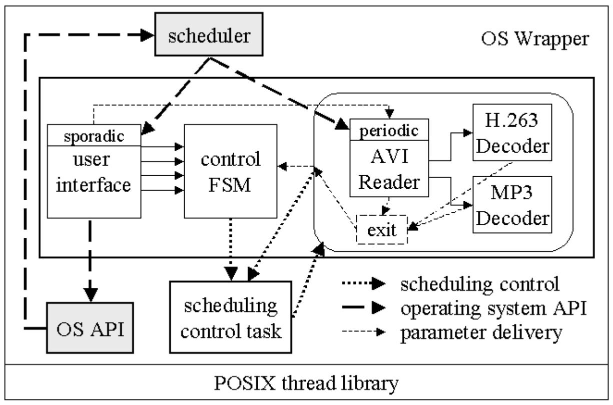
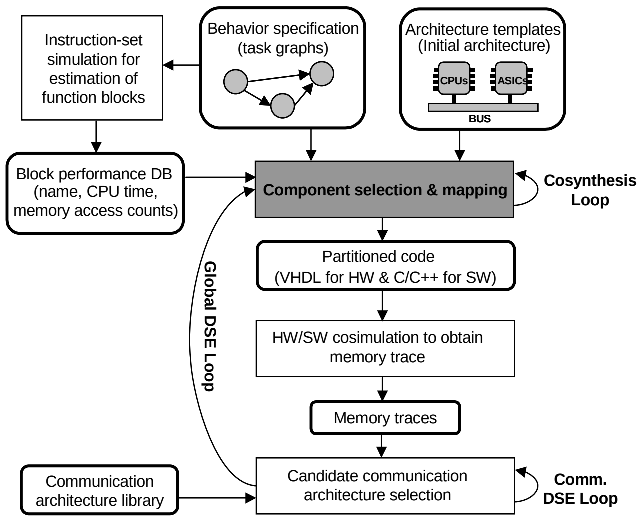


Figure 6. Software structure of the OS wrapper

4. Design space exploration

Figure 7 displays the two-step design space exploration (DSE) loop starting from the initial specification explained in the previous section.



Instruction-set simulation for estimation of function blocks
Behavior specification (task graphs)
Architecture templates (Initial architecture)
CPUs
ASICs
BUS
Block performance DB (name, CPU time, memory access counts)
Component selection & mapping
Cosynthesis Loop
Partitioned code (VHDL for HW & C/C++ for SW)
Global DSE Loop
HW/SW cosimulation to obtain memory trace
Memory traces
Communication architecture library
Candidate communication architecture selection
Comm. DSE Loop

Figure 7. Two-step design space exploration

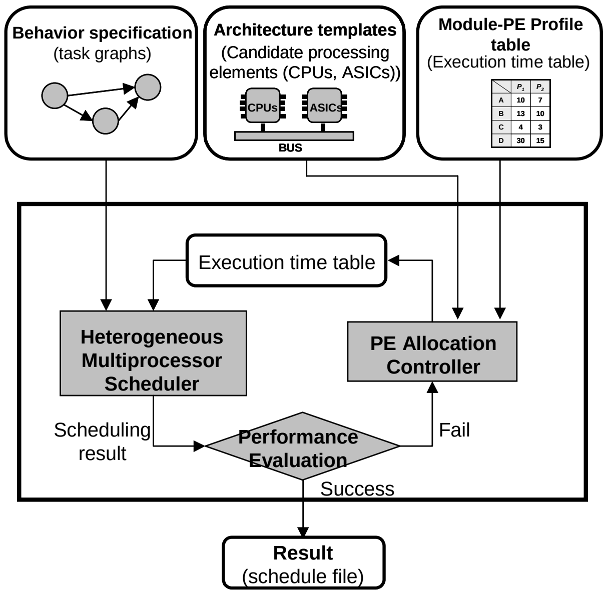
The DSE loop is applied only to the dataflow tasks that are computation intensive while we manually determine which processing element to execute control tasks. By default, the control tasks are mapped to and synthesized for the main control processor of the system. The unique feature of PeaCE as codesign environment is that we maximally utilize the properties of the data flow models at each design step. From the functional specification of system behavior and block performance database, and initial architecture, the cosynthesis loop as the first inner loop of proposed design exploration framework, is performed. The cosynthesis loop solves three sub-problems of cosynthesis: selecting the appropriate processing elements, mapping the function blocks to the selected processing elements, and evaluating the estimated performance or examining the schedulability to check whether the given time constraints are met [11]. In this step we use a very abstract notation of communication overhead by computing it as a product of a fixed cost and the number of data samples. It is because the communication architecture is not determined yet.

After component selection and mapping decision is made, we synthesize the code for each processing component. And we perform a HW/SW co-simulation at the instruction level to obtain the memory trace information from all processing components. Based on the memory trace information and schedule information on the execution order of blocks in each processing element, we perform a communication DSE loop: we first select the candidate communication architectures and then evaluate them using a static queueing analysis and a trace driven simulation. The detailed discussion on the communication DSE loop can be found in [7].

The proposed scheme has a global DSE loop that updates the communication costs used in the cosynthesis loop with those obtained after communication architecture is determined from the communication DSE loop. The key benefits of separating the cosynthesis problem and communication DSE problem are significantly lower time-complexity and extensibility.

4.1 Cosynthesis loop

Figure 8 illustrates the cosynthesis loop whose inputs are a library of candidate processing elements (PEs), a module-PE profile table, and input task graphs.



Behavior specification (task graphs)
Architecture templates (Candidate processing elements (CPUs, ASICs))
CPUs
ASICs
BUS
Module-PE Profile table (Execution time table)
Execution time table
Heterogeneous Multiprocessor Scheduler
PE Allocation Controller
Scheduling result
Performance Evaluation
Fail
Success
Result (schedule file)

	P_1	P_2
A	10	7
B	13	10
C	4	3
D	30	15

Figure 8. Proposed cosynthesis framework

The iteration starts with the PE allocation controller. The PE allocation controller selects a set of processing elements from the input candidate processing elements. Design objectives are considered when selecting the appropriate processing elements. If the design objective is to minimize the cost, we first select the cheapest processor.

The role of the next step is to schedule the acyclic graph of each task to the selected processing elements in order to minimize the schedule length. Since this is a typical problem of heterogeneous multi-processor (HMP) scheduling, we use any scheduler in this step. We apply this step for each task graph separately. This technique assumes that a task uses all hardware resources once scheduled. In case multiple tasks may run concurrently, it would be better to use different processors between tasks. Recently, we devised the "mapping avoidance technique" for that purpose[8]. An interesting observation is that the scheduler may not consume all the selected processing elements.

The next step is performance evaluation step that checks whether the design constraints are satisfied or not. If the schedulability constraint is satisfied, it may end the iteration and record the scheduling results. If any design constraint is violated, it passes the scheduling results and violation information to the PE allocation controller to select other processing elements.

Depending on the operating policy, we apply different methods of schedulability analysis. In case a single task monopolizes all hardware resources, then traditional utilization-based schedulability analysis suffices. But when multiple tasks may run concurrently, we use a schedule-based analysis method assuming Timed-Multiplexing model of operation[8].

4.2 Code generation and trace generation

After mapping is completed, partitioned sub-graphs are constructed for each processing element. At the partitioned boundary, we attach a send block for each output port and a receive block for each input port to describe the interface between processing elements. We should have appropriate send and receive blocks for all candidate target platforms.

When synthesizing a hardware module from a partitioned subgraph, the dataflow semantics should be preserved in the generated code so that the resulting hardware has the identical functional behavior intended by the original graph. And hardware interface logic should be synthesized to support asynchronous interaction with outside modules. Therefore, PeaCE synthesizes HW glue logics and a novel central controller in addition to the RTL codes for the function blocks. After codes are generated, we obtain the memory trace from each processing element through HW/SW co-simulation. The memory trace is used for design space exploration of communication architecture.

4.3 Architecture Optimization

In this step, we seek the optimal bus architecture as well as memory architecture after processing components and hardware platform are determined from the mapping step. Since the design space is huge, we first reduce the design space by queueing analysis of a given bus architecture. From the memory trace information, we extract some statistical parameters to be used in queueing analysis of a given bus architecture. In queueing analysis, we use the static scheduling result of dataflow nodes on each processing element. From our experiments, we confirm that the accuracy of the proposed queueing analysis lies within 10% error bound [7]. For the reduce design space, we use trace-driven simulation to take into account the bus contention and arbitration delay.

5. HW/SW co-verification by co-simulation

After design space exploration loop is exited, we have to verify the design decisions before building an actual prototyping system. In this step, time accurate co-simulation is essential to confirm that the design will satisfy all the design constraints including timing constraints. Seamless CVE of Mentor Graphics Inc. is an example tool that can be used in this step. For time-accurate simulation, we have to run distributed event-driven simulation running multiple simulation processes: for example ARM ISS process and VHDL simulation process. Unfortunately, event driven simulation of multiple component simulators suffers from heavy synchronization overhead, which is not easy to overcome in general.

However, we could increase the simulation speed by reducing the frequency of inter-simulator communications, reducing the active duration of simulators, and utilizing the parallelism of component simulators, without modifying the component simulators. We call the proposed technique "*virtual synchronization*" where local clocks of the component simulators need not be synchronized with the global clock of the co-design backplane, but still managing the correct time stamps of the exchanged event samples [9]. This technique is applicable when task execution results do not depend on the arrival times of input data but their arrival order. Through the virtual synchronization technique, we anticipate two-order magnitude of speed-up of time-accurate co-simulation.

6. Experimental results

The PeaCE codesign environment has been developed as a research prototype system that is designed object-oriented and open-source[1]. It is built on top of the Ptolemy environment that supports hierarchical integration of heterogeneous models of computation[10]. In this section we show two preliminary experimental results based on the proposed design methodology: real-time software synthesis from the proposed system specification, and the HW/SW cosynthesis technique considering the real-time constraints of the tasks.

6.1 Real-time Software Synthesis

In the first experiment, we show the performance result of the automatically generated software of real-time implementation of the Divx player subsystem of the MMMT example. We compare two platforms, Linux kernel 2.6.7 and eCOS_1.3.1. Linux is run on a Pentium processor and eCOS is transplanted on ADS1.2 as an ARMulator simulating ARM922T processor.

At first, we measure the code size overhead of the OS wrapper implementation as shown in Table 1. On the Linux platform, we measured the binary image size of the OS wrapper implementation and that of the total application. The OS wrapper implementation takes about 27KB. Since the most part of this overhead is independent of the number of tasks, the overhead gets less significant as the application size grows. And the overhead for real-time code and for simulation code is comparable in its size. On the eCOS platform, the total application includes the OS wrapper and the POSIX support library as well as eCOS image. The overhead of the OS wrapper implementation is similar to the Linux case in its size. Since we do not need POSIX library for simulation, the last column (Without POSIX) corresponds to the simulation code.

Table 1. Code size overhead of OS wrapper implementation

	Linux		ECOS	
	Real-time code	Simulation code	With POSIX	Without POSIX
OSWrapper/ Total(KB)	27 / 170	25 / 165	23/ 217	22/209

Next, we compared the real-time performance of the generated code by jitter measurements in two different platforms under the same timing properties such as period and frequency of task execution. Jitter is measured by the time difference between the schedule time when the task must wake up and the time on which it actually wakes up. Interrupt dispatch latency affects the jitter. As can be seen in Table 2, jitter in the eCOS platform is much smaller than that in the Linux platform. It is because eCOS platform has small and deterministic interrupt dispatch latency. In the table, T[0] represents the AviParser task, T[1] MP3 decoder task. H263Decoder task is not represented in Table 2 because the H263Decoder task is implemented as a functional task that can be executed immediately after receiving the data samples from the AviParser task.

Table 2. Comparison of jitter in Linux and eCOS

	Linux		eCOS	
Task	T[0]	T[1]	T[0]	T[1]
Period	120	100	120	100
Max_Jitter	1.932	3.362	0.356	0.424
Avg_Jitter	1.434	1.453	0.123	0.090

6.2 Design Space Exploration: Cosynthesis

The second experiment is to show the cosynthesis procedure performed in the PeaCE environment, with a real example, DVR (Digital Video Recorder) system with four channels and a set of randomly generated graphs. We compared the two cosynthesis techniques: original HMP technique and the mapping avoidance technique. The DVR is comprised of four channels. Each channel is made of an H.263 encoder task and has its own real-time constraints: deadline and period. TMN2.0 is used as the reference code for H.263 encoder specification.

As candidate processing elements, we used different versions of ARM processors for programmable processors and an FPGA for hardware processing element. We used 133Mhz ARM720T, 266Mhz ARM920T, 926ej-s, and 399Mhz ARM1020T with L1 cache only. The FPGA as hardware IP core may be used to execute the ME (Motion Estimation), MC (Motion Compensation), DCT, IDCT, and VLC blocks.

Before entering into the cosynthesis framework, we estimated the execution time of blocks on each ARM processor using the ADS (Arm Development Suite) 1.2 on Xeon 2.8GHz dual CPUs for processor simulator and ‘armcc’ compiler included in ADS 1.2 with -02 option for compilation. We assumed that the execution time of hardware IP is about 10 times faster than its software block. The cost of processing elements is assumed proportional to the performance. Since ARM1020T is 4 times faster than the slowest processor, ARM720T, it is assumed 4 times more expensive than it. In this experiment, we used FOREMAN.QCIF as input file.

Experimental results on the schedule length and the partitioning results are shown in Table 3. As shown in Table 3, tighter deadline requires costly resources for both approaches. We used A7, A9, A9e, and A10 as abbreviation for ARM720T, ARM920T, ARM926ej-s, and ARM1020T. The number between parentheses following them indicates the number of used processors to meet the given timing constraint.

The HMP approach needs more expensive processing element earlier than the proposed approach as the deadline is tightened starting from 100 million cycles (about 1.3 frame/sec encoding rate per channel) down to 19 million cycles (about 7.4 frame/sec per channel). And the previous approach prefers a smaller number of faster but expensive processors to a larger number of slower but cheap processors.

Table 3. The partitioning result of 4-channel DVR

Previous		Deadline	Proposed	
Partition result	Cost	(bus cycle)	Cost	Partition result
A9(1)	200	100,000,000 ~ 80,000,000	200	A9(1)
A9(2)	400	70,000,000	400	A9(2)
A10(1)	400	60,000,000	400	A10(1)
A10(1)	400	50,000,000	400	A10(1)
A10(2)	800	40,000,000	800	A9(2), A9e(2)
A10(2), ME	4800	30,000,000	1000	A9(3), A9e(2)
A10(3), ME, DCT, VLC	9200	20,000,000	1800	A10(4), A7(2)
A10(3), ME, DCT, IDCT, VLC	11200	19,000,000	1800	A10(4), A7(2)
N.A.	-	10,000,000	12800	A10(7), MC, DCT, IDCT
N.A.	-	9,000,000	14800	A10(7), MC, ME, DCT, IDCT
N.A.	-	8,000,000	16000	A10(5), MC, ME, DCT, IDCT, VLC
N.A.	-	7,000,000	-	N.A.

The table also shows that the HMP approach converges rapidly to minimum deadline. The minimal schedule length from the previous approach is 18499120 cycles time that represents 7.4 frame/sec per single channel on 133Mhz bus ($(1.8*10^{7})*(7.5*10^{-9}) = 7.4$ sec/frame). It also implies that it is not feasible to use a single OS policy if we want higher encoding rate than 7.4 frame/sec. Thus the table illustrates the design space formed by the processing elements and the operating policies according to the real-time constraints. If we use concurrent execution of four encoding tasks with TM execution policy, we may obtain at maximum 16.9 frame/sec encoding rate per single channel, which means 2.3 times higher performance than the previous approach.

7. Conclusion

PeaCE specifies the system behavior with a heterogeneous composition of three models of computation. The PeaCE environment facilitates all codesign steps from system specification to prototyping utilizing the features of the formal models maximally during the whole design process. Preliminary experiments with real examples prove the viability of the proposed technique.

ACKNOWLEDGMENTS

This work was supported by National Research Laboratory Program (Grant No. M1-0104-00-0015), SystemIC 2010, and IT leading R&D Support Project funded by Korean MIC. The ICT and ISRC at Seoul National University provided research facilities for this study.

References

[1] http://peace.snu.ac.kr/research/peace
[2] S. Edwards, L. Lavagno, E. A .Lee, and A. Sangiovanni-Vincentelli, "Design of Embedded Systems: Formal Models, Validation, and Synthesis," IEEE Proceedings, pp. 366-390, March 1997.
[3] E. Lee, D. Messerschmitt, “Synchoronous data flow”. Proceedings of IEEE, Vol. 75, No. 9, pp. 1235-1245, 1987.
[4] Hyunok Oh and Soonhoi Ha, "Fractional rate dataflow model and efficient code synthesis for multimedia applications", ACM SIGPLAN Notice Vol. 37 July 2002, pp 12-17.
[5] Chanik Park, Jaewoong Chung and Soonhoi Ha, "Extended Synchronous Dataflow for Efficient DSP System Prototyping", Design Automation for Embedded Systems, Kluwer Academic Publishers Vol. 3 March 2002, pp 295-322.
[6] Dohyung Kim, Soonhoi Ha, "Static Analysis and Automatic Code Synthesis of flexible FSM Model", ASP-DAC 2005 Jan 18-21 2005.
[7] Sungchan Kim, Chaeseok Im, and Soonhoi Ha, "Efficient exploration of on-chip bus architectures and memory allocation", CODES+ISSS, pp 248-253 , Sep. 2004.
[8] Choonseung Lee and Soonhoi Ha, "Hardware-Software Cosynthesis of Multitask MPSoCs with Real-Time Constraints", The 6th International Conference on ASIC, Oct. 24-27 2005
[9] Dohyung Kim, Chan-Eun Rhee, and Soonhoi Ha, "Combined Data-driven and Event-driven Scheduling Technique for Fast Distributed Cosimulation," IEEE Transactions on Very large Scale Integration(VLSI) Systems Vol. 10 pp 672-679 October 2002
[10] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," Int. Journal of Computer Simulation, special issue on Simulation Software Development, vol. 4, pp. 155-182, April, 1994.
[11] H. Oh and S. Ha, "Hardware-Software Cosynthesis of Multi-Mode Multi-Task Embedded Systems with Real-Time Constraints", Codesign Symposium (CODES), 2002.
[12] D. Verkest, et. al., "CoWare - a design environment for heterogeneous hardware/software systems," Design Automation for Embedded Systems, 1(4), Oct. 1996.