Automatic Generation Including Fast Timed Simulation Models of Operating Systems in Multiprocessor SoC Communication Design

Sungjoo Yoo Gabriela Nicolescu Lovic Gauthier Ahmed A. Jerraya

SLS Group, TIMA Laboratory
46 Avenue Félix Viallet, 38031 Grenoble, France
{Sungjoo.Yoo,Gabriela.Nicolescu,Lovic.Gauthier,Ahmed.Jerraya}@imag.fr

Abstract

To enable fast and accurate evaluation of HW/SW implementation choices of on-chip communication, we present a method to automatically generate timed OS simulation models. The method generates the OS simulation models with the simulation environment as a virtual processor. Since the generated OS simulation models use real OS code, the presented method can mitigate the OS code equivalence problem. The generated model also simulates different types of processor exceptions. This approach provides two orders of magnitude higher simulation speedup compared to the simulation using instruction set simulators for SW simulation.

1 Introduction

Communication refinement is a crucial design step in System-on-Chip (SoC) design since it has significant impact on the performance of implemented SoCs in terms of power, runtime, area, etc. [1]. Communication refinement consists of two steps: communication network design and wrapper design. Communication network can be on-chip buses [2]. circuit switch networks, packet switch networks, etc. Wrappers are required for communication between applications running on different processors, DSPs, IPs, etc. via communication networks. Wrappers are constructed by software (SW) in the form of operating systems (OS's) [3] as well as in the form of hardware (HW) [4][5].

Due to the constraints (in terms of performance, power, area, etc.) given to the embedded SoCs, the design of communication networks and wrappers needs to be optimized. For instance, communication network topologies or parameters (e.g. bus priorities, DMA sizes, slot assignments in TDMA-style buses, etc.) need to be determined to optimize the performance of SoC design [6][2]. Wrapper designs can also be optimized. For instance, the size of embedded OS can be scalable or minimized [7][3].

In terms of design space in such an optimization, there are huge numbers of design alternatives. Thus, to obtain practically optimal designs of communication networks and wrappers, design space exploration (DSE) should be fast enough to meet the given tight time-to-market. To obtain fast DSE, a fast evaluation of design alternative is necessary. Due to the complex behavior related to on-chip communication (e.g. bus conflicts, task scheduling effects, etc.), we need also accurate evaluation.

As the evaluation method, we run simulation. During DSE of on-chip communication, the designer determines a design alternative, builds a corresponding simulation model, then runs simulation to evaluate the performance of the selected design alternative. In the above process, to obtain fast evaluation, we need automatic building of simulation models as well as fast simulation.

To run fast simulation of HW implementation of on-chip communication, we can run high-speed cycle-accurate simulation models that can also be synthesizable (e.g. synthesizable C) [8][9] or fast simulation models of on-chip communication networks. For the SW implementation of on-chip communication, i.e. OS's, instruction set simulators (ISS's) can be run, where the OS's and the application SW are simulated. However, to achieve fast design space exploration by fast evaluation, the speed of ISS's can be prohibitively slow. Thus, fast and accurate simulation models of OS's are required.

For OS simulation models, in previous work, there are three types of OS simulation model: native OS [10], virtual OS [11][12] and aggregate timing model of OS [13]. Details of previous work will be explained in Section 2. To clarify our contribution, in comparison with virtual OS concept, ours gives OS simulation models based on virtual processor concept thereby reducing the gap between OS simulation model and real OS code. Compared with the aggregate timing model of OS, ours gives finer grain delay models. The previous work lacks in automatic building of OS simulation models (in [11][12]), accurate timed OS simulation (in [10]), and/or code equivalence between real OS code and OS simulation model (in [11][12][13]). In our work, we present a method that automatically generates timed OS simulation models with real OS codes.

This paper is organized as follows. Section 2 presents related work. Section 3 explains our flow of communication refinement and simulation model generation. Section 4 addresses automatic generation of fast timed OS simulation models. Section 5 gives experiment results. Section 6 concludes the paper.

2 Previous Work

2.1 Three Types of Timed Simulation of Embedded SW

We classify timed simulation of embedded SW into three types: (1) functional simulation (with delay annotation), (2) usage of OS simulation models, and (3) usage of instruction set simulators.

Functional simulation of embedded SW uses the simulation environment (e.g. SystemC) for scheduling of SW tasks (e.g. by event notification) and inter/intra-processor communication between SW tasks (e.g. by signals). In this case, timing is simulated mainly for SW applications while the timing delays of scheduling SW tasks or communication between them are not accurately simulated.

OS simulation models will be explained in detail in the following subsection. Instruction set simulation enables more accurate (e.g. instruction/cycle/phase-accurate) simulation of SW (including SW applications and OS) running on the processor.

2.2 OS Simulation Models

There are three types of OS simulation model: native OS, virtual OS, and aggregate timing model of OS. In the followings, we explain each of the three types and compare them in terms of real OS code usage, timed OS simulation and automatic generation of OS simulation model.

2.2.1 Native OS

Native OS runs on the simulation host simulating the real OS. For instance, WindRiver Systems Inc. provides VxSim as a native simulation model of its RTOS, VxWorks [10]. The purpose of native OS is to validate the functionality of SW applications running on the OS. When designing multiprocessor SoCs, multiple native OS's can run concurrently. They communicate with each other using

interprocess communication (e.g. Unix sockets, pipes, etc.) supported by the simulation hosts (e.g. workstations). However, they lack in modeling the HW part that surrounds the real processors on which the OS's run. Thus, timed cosimulation between multiple OS simulation and HW simulation is not usually supported. Support of real code usage and automatic generation of OS simulation models depends on OS vendors.

2.2.2 Virtual OS

The virtual OS simulates the functionality of real OS. The main purpose of using virtual OS's is to validate the functionality and timing of design decisions of OS implementation.

CarbonKernel provides a tool for the designer to develop OS simulation models based on a basic virtual RTOS [12]. SoCOS also enables to model real OS's with a generic OS simulation model [11]. In both cases, the designer can add timing delays of code sections into the virtual OS. The virtual OS can be applied to various types of OS's as far as the designer designs the simulation models specific to different OS's and adds them to the virtual OS.

The virtual OS has a problem which we call a *code equivalence problem*. This originates from the fact that the code of virtual OS is not equal to that of real OS. For instance, the task scheduler code of virtual OS cannot be exactly the same to that of a specific OS that the designer uses or designs for himself. Thus, to fully validate the functionality of real OS, the designer needs to run more accurate simulation such as running instruction set simulators.

Another problem of virtual OS is that it is not flexible to try numerous candidates of OS implementation. When using the virtual OS, to obtain an OS simulation model specific to a candidate of OS implementation, the designer needs to do "personalization" of virtual OS [12]. In this case, personalization is to add/modify (new) functionalities of candidate OS implementation to/in the virtual OS. Such a process is usually manual, time-consuming and error-prone. Thus, manual personalization cannot enable fast DSE in obtaining optimal OS implementations.

2.2.3 Aggregate Timing Model of OS

The aggregate timing model of OS is to simulate the timing delay of OS in an aggregate delay model. For instance, a task scheduling delay or a context switch delay can be calculated as a function of the number of ready tasks or the size of task context. Then, the delay is counted when the task invocation or context switching is simulated in timed simulation of embedded SW.

This model has been used in the area of real-time systems, especially to model the effect of task scheduler in task schedulability analysis. The aggregate delay is usually measured from the execution of real processor on a system board or from the simulation done by RTOS vendors.

In aggregate delay models, the timing accuracy of simulation may not be satisfactory. Another drawback of this model is poor flexibility. To enable DSE in OS optimization, the aggregate delay model needs to be applicable to each of OS implementation candidates (e.g. customized task scheduling policies). However, current practice of measuring delays (by simulation or measurement) cannot be flexible enough to give fast calculation of delays in specific OS implementations.

In the review of previous work, new OS simulation models need to support (1) real OS code (as much as possible to mitigate the code equivalence problem), (2) timed simulation, and (3) automatic generation of OS simulation models.

3 Communication Refinement of Multiprocessor SoCs

Figure 1 shows our flow of communication refinement from the system specification at *macro-architecture level* to the SoC implementation at *micro-architecture level*. The macro-architecture specification consists of modules and channels. A module consists

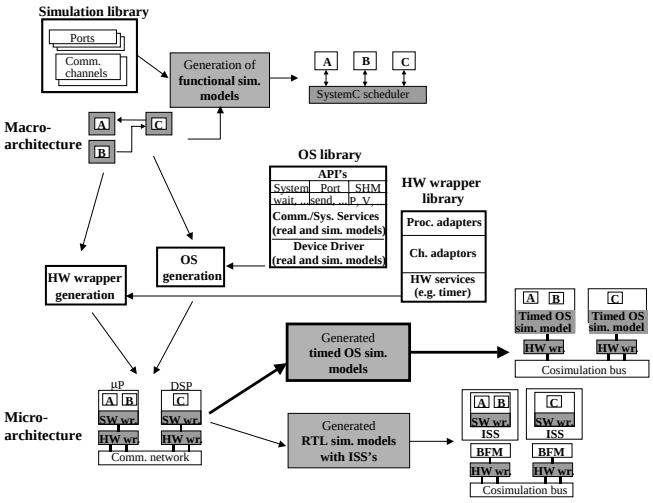

Figure 1: Micro-architecture generation flow.

of behavioral part and ports. The behavioral part requests communication and system services to its external world via ports. Channels provide ports with *communication services* such as FIFO's, semaphores, registers, etc. Ports itself can provide the behavioral part of module with *system services* such as timers, scheduling, exception handling, etc. The designer writes the macro-architecture specification and then validates the functionality by automatically generating the functional simulation models at macro-architecture level.

To generate the micro-architecture implementation, conventionally, RTL implementation, two types of wrapper are generated: HW and SW wrappers. The communication and system services provided by the communication channels and ports are implemented by HW and SW wrappers and the communication network at micro-architecture level. In Figure 1, for instance, two modules, A and B in the macro-architecture specification have been mapped on a μP in the micro-architecture implementation. Two macro-architecture communication channels (arrows in the figure) between modules A, B, and C are implemented on HW and SW wrappers and a communication network at micro-architecture level.

In terms of implementation, the HW wrapper is a processor interface that connects the processor to the communication network at micro-architecture level (for further details, refer to [5]). The SW wrapper is an OS that enables the application SW to perform inter/intra-processor communication (for further details, refer to [3]). To generate HW and SW wrappers, two libraries are used: OS library and HW wrapper library (Figure 1). According to the design decisions made by the designer, the wrapper generation can give different implementations of SW and HW wrappers. Thus, automatic wrapper generation can enable the designer to try alternative design choices in communication refinement.

The generation of HW and SW wrappers includes generation of simulation models as well as synthesizable codes [14]. Thus, in the OS and HW wrapper libraries, there are simulation models as well as synthesizable codes. At micro-architecture level, the designer can perform two types of simulation: cycle-accurate simulation with instruction set simulators (ISS's) and cycle-approximate simulation with timed OS simulation models. In Figure 1, two types of simulation are exemplified. In this paper, we present a method to generate timed OS simulation models (the path shown in the figure with bold arrows).

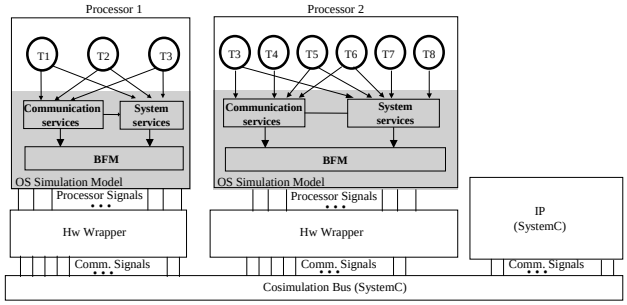

Figure 2: Timed cosimulation models of VDSL modem application.

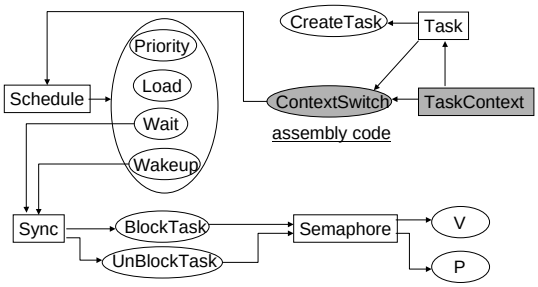

Figure 3: An example of service dependency.

4 Automatic Generation of Fast Timed OS Simulation Models

4.1 Basic Strategies and Requirements

In generating timed OS simulation models, our basic strategies are

- OS simulation with real OS code with delay annotation and without using ISS's
- Generation of OS simulation models is generating the real OS with the simulation environment as a virtual processor target.

To run OS simulation, the simulation environments are required to support (1) *processes* and *events* and (2) *dynamic sensitivity*. We map a process of simulation environment to each of SW tasks running on the OS. Dynamic sensitivity is to be able to change the sensitivity list of process during run time. It is necessary to simulate the preemption and resumption of task execution.

Figure 2 shows a micro-architecture level simulation model of a VDSL application that we use in our experiments. In the figure, two OS simulation models are shaded, one for each of two processors. An OS simulation model simulates communication and system services and includes a bus functional model (BFM) of processor.

4.2 Automatic Generation of OS Simulation Models

To generate OS simulation models, we use the same method used to generate/configure real OS codes [3]. The basic idea of this method is to find the OS services that are required by the application SW and then to generate their codes according to the target processor. Figure 3 exemplifies how to find required OS services.

In the figure, ovals represent OS services (and their codes) and rectangles code sections related to the OS services. Arrows represent relationship between service/code provider and requester. For instance, scheduling services, Priority, Load, Wait, and Wakeup use code section Schedule and service ContextSwitch. Such a relationship can be transitive. For instance, semaphore services P and V use the four scheduling services through services BlockTask and UnblockTask and code sections Sync and Semaphore. Thus,

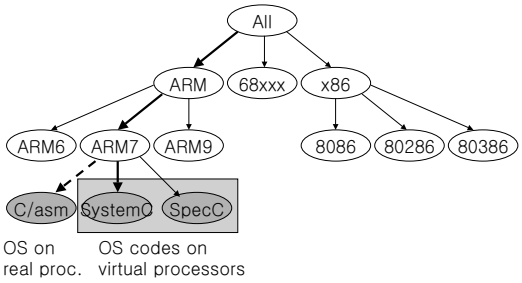

Figure 4: An example of processor dependency.

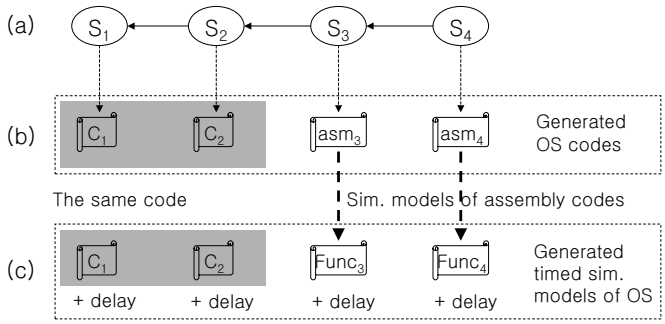

Figure 5: Examples of OS codes and simulation models.

if semaphore services P and V are used by the application SW, according to the dependency chain shown in the figure, all the codes of services and related code sections shown in the figure are required to be included into the OS to be generated.

When generating the codes of required services, they can be high-level codes such as C or low-level code such as assembly code. In the figure, the codes of TaskContext and ContextSwitch are denoted as assembly codes. OS codes can be processor-dependent (e.g. assembly codes or processor-dependent C codes) or processor-independent (e.g. normal C codes).

Based on the dependency of services, the OS generation process performs a composition of the corresponding code sections as explained in [3].

As mentioned in section 4.1, generation of OS simulation models is generating the real OS with the simulation environment as a virtual processor target. Figure 4 shows how to treat simulation environments as virtual processor targets. The figure shows a tree relationship in the codes corresponding to each OS service. In the figure, ovals represent code sections that are processor-independent (denoted by All) or processor-dependent (denoted by the names of processors). For instance, OS code sections of the same service can be different depending on ARM or 68xxx processor, or ARM6, ARM7, and ARM9 in the ARM processor family. If the designer determines ARM7 as the target processor, corresponding C or assembly codes of ovals All, ARM, and ARM7 are used to generate the code of each of OS services. Note that such a tree relationship is applied to the code of each of OS services, independently.

To generate OS simulation models, we use another level in the figure. For instance, if we have two simulation environments, SystemC and SpecC, then the oval ARM7 has three child ovals, C/asm, SystemC, and SpecC. That is, OS simulation codes specific to the simulation environments are prepared with the simulation environments as virtual processors and used to generate OS simulation models. Then, if the designer needs to generate the OS code for implementation, the code section denoted C/asm is used to generate the OS. If he/she needs to simulate the real OS code on a simulation environment, for instance, SystemC, then the simulation code of oval SystemC is used to generate the OS code. In this case, the generated OS code is the OS simulation model that runs on the selected simulation environment.

Figure 5 shows the comparison of generating the OS code and the OS simulation model. In Figure 5 (a), four ovals are OS services, solid arrows represent service provider and requester relationship, and dashed arrows correspondence between services and codes. In the generated OS code, as shown in Figure 5 (b), for two OS services S_1 and S_2, two C codes C_1 and C_2 are used, for the other two OS services S_3 and S_4, two assembly codes asm_3 and asm_4 are used. In the generated OS simulation model, for S_1 and S_2, assuming the C code is processor-independent, the same C codes C_1 and C_2 are used with delay annotation. For S_3 and S_4, their functional simulation models $Func_3$ and $Func_4$ are used to simulate their functionality with delay annotation. Details of functional simulation models will be given in section 4.4.

As shown in the figure, by using the same codes (C_1 and C_2 in this example) in both the real OS and the OS simulation model, the code equivalence problem of virtual OS can be mitigated in our method. In the real OS code, the assembly code constitutes less than 5% of the total code size. Thus, more than 95% of our OS simulation model can be the real OS code.

4.3 Timed OS Simulation Models

4.3.1 Timed RPC Process

In OS simulation, to have a single thread of execution of each SW task, communication and system services and BFM are called via RPC's (remote procedural calls).

For timed simulation of OS as well as the application SW, we use a function called delay to add delay annotation into the code. In our simulation implementation, we use a global clock in both OS simulation models and HW simulation involved in multiprocessor SoC simulation. Thus, the delay function synchronizes SW and HW simulation.

To use the delay function in any SW task codes, services, and the BFM, RPC functions should also enable the simulation time to advance. We call such an RPC function a *timed RPC process*. The implementation of timed RPC process depends on the simulation environment. Our implementation will be given in section 5.

4.3.2 Simulation of Processor Exception Handling

To simulate the timing behavior of OS, modeling processor exception handlers is necessary. Processors can have several types of processor exception. For instance, ARM processor has seven different types of exception: reset, undefined instruction, software interrupt (SWI), prefetch abort, data abort, IRQ, and FIQ [15].

We observe that to validate the functionality and performance of communication refinement, all the exceptions are not required to be modeled. For instance, the exception of undefined instruction is not related to the communication refinement. In the case of ARM processor, three exceptions, SWI, IRQ, and FIQ are related to communication refinement. Thus, they are required to be modeled for OS simulation.

To show how to model exception handlers in OS simulation, an example of SWI code of ARM7 processor is shown in Figure 6. The figure shows a SWI routine in assembly code (_SWI_Routine) and a C code section to call a generic SWI function called __trap_trap with SWI number 0 (defined by __swi(0)). When the function in line 17 is called, the processor execution jumps to the vector table element of SWI, then the SWI handler in line 2 is executed.

To model such exception handlers, our strategy is to model the minimal set of elements to model them to have fast OS simulation. In the case of ARM processor, the minimal set of elements is made of processor mode registers (CPSR and SPSR's) that contain control bits such as interrupt masks specific to each processor mode.

Figure 7 shows a model of SWI routine for OS simulation and their usage in C code to simulate the SWI call. Two functions SWI_Enter and SWI_Return model the entry and return operations of SWI routine. For instance, the function SWI_Enter corresponds

```
1   // assembly code for SWI routine
2   _SWI_Routine
3   STMIA  r13,{r0-r14}^   ; Push USER registers
4   MRS    r0,spsr         ; Get spsr
5   STMDB  r13!,{r0,lr}    ; Push spsr and lr_svc
6   LDR    r0,[lr,#-4]     ; Load swi instruction
7   BIC    r0,r0,#0xff000000
8   BL     __trap_trap
9   LDMIA  r13!,{r0,lr}    ; Pop return address and spsr
10  MSR    spsr_cf,r0      ; Restore spsr for swi
11  LDMIA  r13,{r0-r14}^   ; Restore registers and return to user mode
12  NOP                    ; NOP
13  MOVS   pc,lr           ; Return from SWI
14
15  // C code to use SWI
16  __swi(0) void __trap_trap(int, int, int);
17  __trap_trap(0, id, 0);
```

corresponding to SWI_Enter

Figure 6: SWI routine: assembly and C code parts.

```
// Counterparts of SWI enter and return
SWI_Enter() {
        CPSR_save = CPSR;
        SPSR_save = SPSR;
        CPSR = SVC;
}

SWI_Return() {
        CPSR = CPSR_save;
        SPSR = SPSR_save;
}

// Counter part of C code
SWI_Enter() ;
__trap_trap(0,id,0);
SWI_Return();
```

Figure 7: Simulation model of SWI routine.

to the code section, line 3 to line 7 in Figure 6. In the functions, only the change of mode registers (CPSR and SPSR) is simulated. In the C code that calls SWI, each of the two functions is added before and after the SWI call shown in Figure 7. We model the other exception handlers such as HW interrupt handlers in the same way.

To model HW interrupts, we insert a function sync_int where we need to simulate the preemption of task execution by HW interrupts. The function of sync_int is to check the values of interrupt pins (nIRQ and nFIQ in the case of ARM processor) to see if a new interrupt arrives. If there is a new interrupt, the simulation model of interrupt service routines corresponding to the interrupt is called. If not, the function sync_int just returns without advancing the simulation time.

The frequency of calling sync_int can determine the timing accuracy of simulating HW interrupt handling. However, too frequent execution of sync_int can also degrade simulation performance. Thus, in our simulation flow, the designer can locate sync_int functions by trading off between simulation performance and accuracy.

In terms of modeling task preemption, [16] presents a method to model interrupt handling. In the work, processor modes are not separately modeled and it is assumed that the order of task execution does not change by the interrupt handling. In our modeling method, ISRs can call task schedulers to invoke new tasks before returning to the task execution preempted by the interrupt.

4.4 Simulation Models of Processor-Dependent Codes

Examples of processor-dependent codes are boot code, task state code, context switch code and exception handlers. For such processor-dependent assembly codes, we use their functional models with performance annotation (using delay functions). Figure 8 shows an example of assembly code and a simulation model of context switch code of ARM processor. In the figure, a list of events, wakeup_event is used to suspend (by wait function) and to resume (by notify function) task execution. We simulate also the execution delay of context switch using the delay functions.

The boot code sets vector tables, stack ranges, etc. We use a behavioral model of the boot code that is simulated at the initialization (i.e. the constructor function) of the OS simulation model. At the beginning of simulation, to serialize the execution of SW tasks,

```
__cxt_switch          ;r0, old stack pointer, r1, new stack pointer
STMIA   r0!,{r0-r14}    ; save the registers of current task
LDMIA   r1!,{r0-r14}    ; restore the registers of new task
SUB     pc,lr,#0        ; return
END
```

(a) Context switch: assembly code

```
RPC_Process context_sw(int cur_task_id, int new_task_id)
{
        wakeup_event[new_task_id].notify(); delay(50);
        wait(wakeup_event[cur_task_id]); delay(40);
}
```

(b) Context switch: simulation model

Figure 8: Context switch code and simulation model of ARM processor.

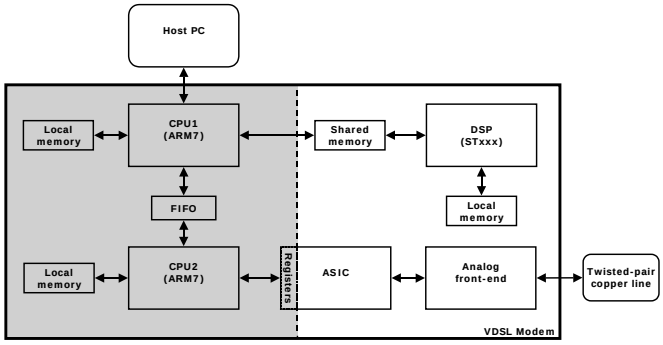

Figure 9: VDSL modem application.

each task suspends its execution by waiting for the synchronization event (i.e. wakeup_event) coming from the task scheduler service. In our OS simulation models, since we do not simulate the actual processor, task states such as registers and stacks are not simulated. One can advocate that for this part we have a code equivalence problem. This is true, however since this is generally less than 5% of the full OS code, the code equivalence problem is significantly mitigated.

4.5 Timing Calculation and Application to Configurable OS's

To calculate the execution delay values used in delay functions, we can use conventional estimation methods of SW execution time [13][17][18]. To have processor-dependent delay values, before the OS simulation model is generated, the delay values are calculated for the target processor and then included in the OS simulation model.

To apply the automatic generation of timed OS simulation model to configurable OS's, the required steps are as follows.

- Add timing delays to the existing codes of OS.
- Prepare the simulation models of processor-dependent codes.
- Apply the automatic OS generation/configuration flow in [3] with the simulation environment as a virtual processor target.

5 Experiments

We applied the presented method to the design of a VDSL modem design as shown in Figure 9. The VDSL modem uses Discrete Multi-Tone (DMT) modulation.[1] We design a part of the system with two ARM7 processors. The part we design as a multiprocessor SoC is shown on the left of Figure 9, inside the gray box. The VDSL core functions, the analog interface, and the DSP core are implemented in a third-party block. The DSP and the ASIC block execute functions such as (I)FFT, Reed-Solomon (de)coding, (de)scrambling, and (de)framing.

To configure, monitor and synchronize the DSP and the ASIC block we map the control tasks, the host interface tasks, and the high-level VDSL code into two ARM7 processors (CPU1 and CPU2 in Figure 9). CPU1 runs three concurrent SW tasks and CPU2 runs six concurrent SW tasks.

To generate OS's, we use our OS generation tool [3]. As simulation environment, in our experiments, we use SystemC [19]. We map a SW task to a thread in SystemC. Since SystemC provides dynamic sensitivity in the member functions of module, we map an RPC process to a member function of OS model in SystemC.

Refining the VDSL application down to micro-architecture level implementation, we generate the simulation models of HW wrappers and OS's as shown in Figure 2. We run two types of simulation: one using two ISS's (one for each ARM processor) for SW simulation and the other using the generated OS models for SW simulation. For the other HW parts, we use the same simulation models in SystemC.

In our experiments, the generated OS simulation models give more than two orders of magnitude higher simulation speedup compared to the use of ISS's. When the number of ISS's is larger than in our case (two ISS's in our case), this speedup will be even larger due to the synchronization overhead between multiple ISS's. We will also investigate the effects of frequent calls of sync_int in terms of simulation runtime and simulation accuracy.

6 Conclusion

In this paper, we presented a method of automatic generation of timed OS simulation models. Automatic generation of OS simulation models will enable the designer to try more design alternatives of OS implementation during design space exploration of communication refinement. Since the generated OS simulation models contain real OS codes, compared to virtual OS approaches, our method mitigates the OS code equivalence problem. The simulation speedup in experimental results shows the effectiveness of our method.

[1] DMT uses 247 Quadrature Amplitude Modulated (QAM) carriers on channels of 4kHz bandwidth, and a total bandwidth of 11.04 MHz.

References

[1] K. Lahiri, A. Raghunathan, G. Lakshminarayana, and S. Dey, "Communication Architecture Tuners: A Methodology for the Design of High-Performance Communication Architectures for System-on-Chips", *Proc. Design Automation Conf.*, pp. 513–518, June 2000.

[2] Sonics, Inc., "Silicon Backplane μNetwork", available at http://www.sonicsinc.com/Pages/Networks.html.

[3] L. Gauthier, S. Yoo, and A. A. Jerraya, "Automatic Generation and Targeting of Application Specific Operating Systems and Embedded Systems Software", *Proc. Design Automation and Test in Europe*, Mar. 2001.

[4] C. K. Lennard, P. Schaumont, G. de Jong, A. Haverinen, and P. Hardee, "Standards for System-Level Design: Practical Reality or Solution in Search of a Question?", *Proc. Design Automation and Test in Europe*, pp. 576–585, Mar. 2000.

[5] D. Lyonnard, S. Yoo, A. Baghdadi, and A. A. Jerraya, "Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip", *Proc. Design Automation Conf.*, June 2001.

[6] K. Lahiri, A. Raghunathan, and S. Dey, "Efficient Exploration of the SoC Communication Architecture Design Space", *Proc. Int'l Conf. on Computer Aided Design*, pp. 424 – 430, Nov. 2000.

[7] E. Verhulst, "From a distributed embedded RTOS to a programatic framework for multi-core SoC", *MP SoC school, Aix-les-Bains, France*, July 2001.

[8] Synopsys, Inc., "CoCentric SystemC Compiler", available at http://www.synopsys.com/.

[9] Coware, Inc., "N2C", available at http://www.coware.com/cowareN2C.html.

[10] WindRiver Systems, Inc., "VxWorks 5.4", available at http://www.wrs.com/products/html/vxwks54.html.

[11] D. Desmet, D. Verkest, and H. De Man, "Operating System Based Software Generation for Systems-on-Chip", *Proc. Design Automation Conf.*, June 2000.

[12] "CarbonKernel", available at http://www.carbonkernel.org/.

[13] Cadence Design Systems, Inc., *Virtual Component Codesign*.

[14] S. Yoo, G. Nicolescu, D. Lyonnard, A. Baghdadi, and A. A. Jerraya, "A Generic Wrapper Architecture for Multi-Processor SoC Cosimulation and Design", *Proc. Int'l Workshop on Hardware-Software Codesign*, 2001.

[15] D. Jaggar, *Advanced RISC Machines Architectural Reference Manual*, Prentice Hall, July 1996.

[16] J. Cockx, "Efficient Modeling of Preemption in a Virtual Prototype", *Proc. IEEE International Workshop on Rapid System Prototyping*, June 2000.

[17] M. Lajolo, M. Lazarescu, and A. Sangiovanni-Vincentelli, "A Compilation-based Software Estimation Scheme for Hardware/Software Co-Simulation", *Proc. Int'l Workshop on Hardware-Software Codesign*, May 1999.

[18] K. Suzuki and A. Sangiovanni-Vincentelli, "Efficient Software Performance Estimation Methods for Hardware/Software Codesign", *Proc. Design Automation Conf.*, June 1996.

[19] Synopsys, Inc., "SystemC, Version 2.0", available at http://www.systemc.org/.