Automatic Generation of Fast Timed Simulation Models for Operating Systems in SoC 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 final 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 consumption, runtime, area, etc. [1]. Communication refinement consists of two steps: communication network design and wrapper design. Communication network can be on-chip buses [2][3], circuit switch networks [4], packet switch networks [5][6], etc. Wrappers are required to adapt modules to communication networks. Wrappers are constructed in the form of software (SW), i.e. operating system (OS)[1] [7] as well as in the form of hardware (HW) [8][9].

Due to the constraints (in terms of performance, power, area, etc.) given to the embedded SoCs, the design of wrappers needs to be optimized. For instance, the size of embedded OS can be scalable or minimized [7][11]. Communication network topologies or parameters (e.g. bus priorities, DMA sizes, slot assignments in TDMA-style buses, etc.) need also to be determined to optimize the performance of SoC design [3][10].

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.

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) [12][13] or fast simulation models of on-chip communication networks. For the SW implementation, instruction set simulators (ISSs) can run the OSs and the application SW. However, to achieve fast design space exploration by fast evaluation, the speed of ISSs can be prohibitively slow. Thus, fast and accurate simulation models of OSs and software are required.

For OS simulation models, in previous work, there are three types of OS simulation model: native OS [14], virtual OS [15][16] and aggregate timing model of OS [17]. Details of previous work will be explained in Section 2. Our contribution, is close to virtual OS concept, it uses OS simulation models based on virtual processor concept. This reduces the gap between OS simulation model and final 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 [15][16]), ac-

[1] In this paper, the operating system includes device drivers.

curate timed OS simulation (in [14]), and/or code equivalence between final OS code and OS simulation model (in [15][16][17]). 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 and inter/intra-processor communication between SW tasks based on events and/or 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 final OS code usage, timed OS simulation and automatic generation of OS simulation model.

2.2.1 Native OS

Native simulation runs the OS on host workstation. For instance, WindRiver Systems Inc. provides VxSim as a native simulation model of its RTOS, VxWorks [14]. The purpose of native OS simulation is to validate the functionality of SW applications running on the OS. When designing multiprocessor SoCs, multiple native OSs 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 OSs run. Thus, timed cosimulation between multiple OS simulation and HW simulation is not usually supported. Support of final 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 a real OS. The main purpose of using virtual OSs 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 [16]. SoCOS also enables to model final OSs with a generic OS simulation model [15]. 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 OSs as far as the designer designs the simulation models specific to different OSs and adds them to the virtual OS.

The virtual OS suffers from the *code equivalence problem*. This originates from the fact that the code of virtual OS is different from final 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 final OS, the designer needs to run more accurate simulation such as using instruction set simulators.

Another problem of virtual OS is that it is not flexible and do not allow to try numerous candidates of OS implementation. To simulate a specific OS implementation, the designer needs to do “personalization” of the virtual OS [16]. 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 and 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

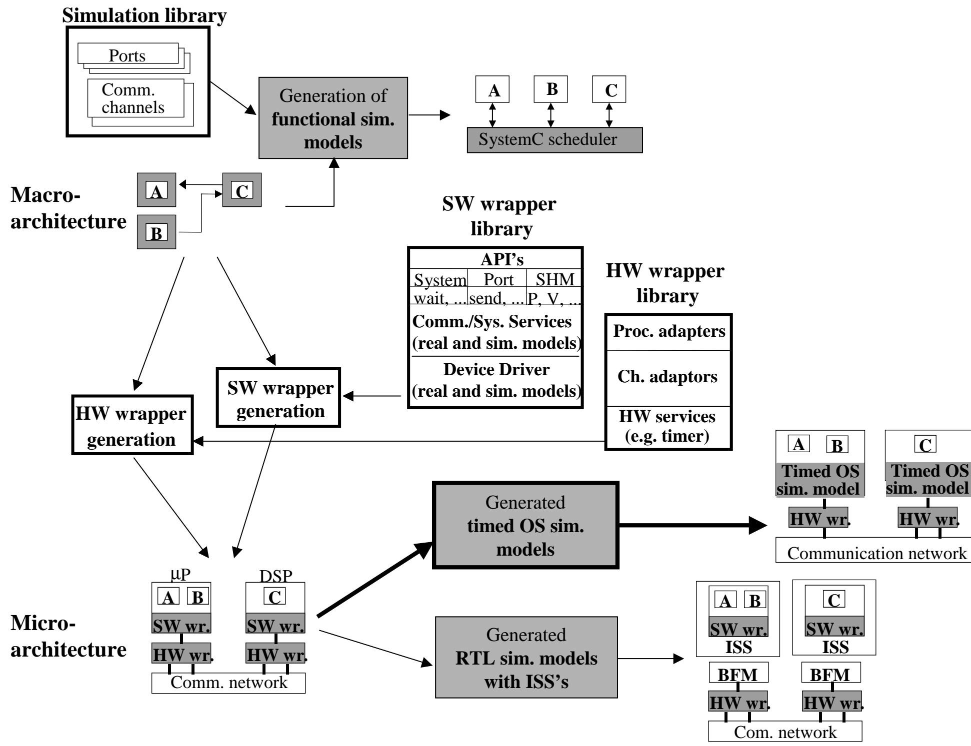

Figure 1. Micro-architecture generation flow.

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) final 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. In the figure, three modules (A, B, and C) and two channels (arrows) are exemplified. A module consists of behavioral part (shown as blank rectangles in the figure) and ports. The behavioral part requests communication and system services to its external world via ports. Channels provide ports with *communication services* such as FIFOs, semaphores, registers, etc. Ports themselves 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.[2] The communication and system services provided by the communication channels and ports are implemented by HW and SW wrappers and by 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. The two macro-architecture communication channels 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 [9]). The SW wrapper is an OS that enables the application SW to perform inter/intra-processor communication (for further details, refer to [7]). To generate HW and SW wrappers, two libraries are used: SW and HW wrap-

[2] In this paper, we use two terms, OS and SW wrapper, interchangeably.

per libraries (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 [18]. Thus, in the SW 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 (ISSs) 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).

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 final OS code with delay annotation and without using ISSs
- Generation of OS simulation models is generating the final 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 (Very high bit-rate Digital Subscriber Line) 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 Application-Specific OSs

To generate OS simulation models, we use the same method used to generate/configure final OS codes [7]. The basic idea of this method is to find the OS services that are

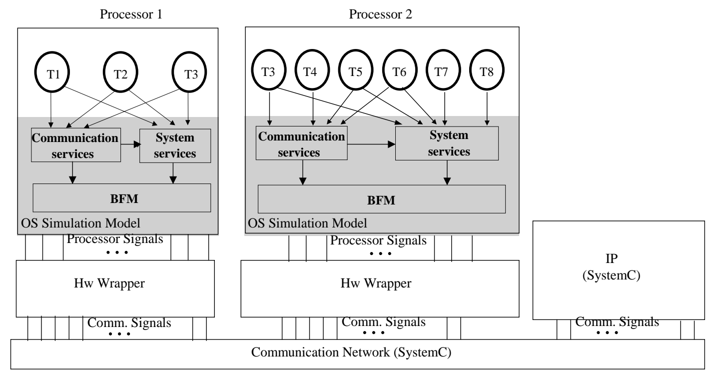

Figure 2. A timed cosimulation model of VDSL modem application.

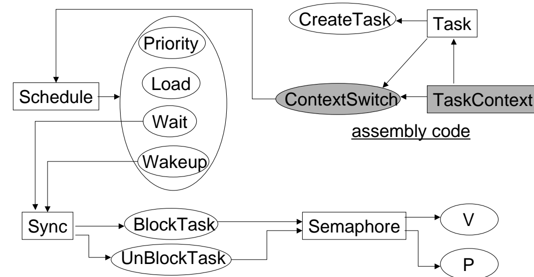

Figure 3. An example of service dependency.

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, 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 the codes of required services are generated, 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

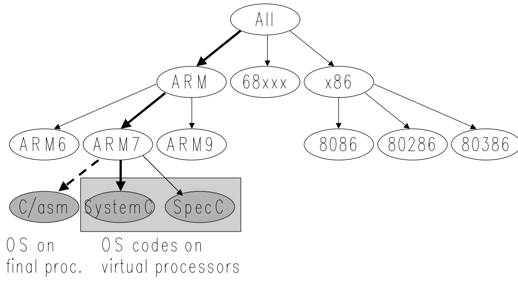

Figure 4. An example of processor dependency in the OS codes.

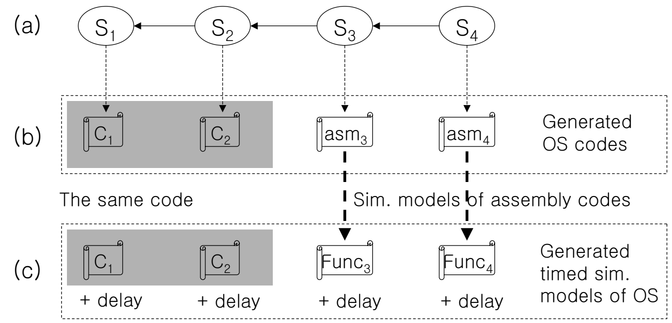

Figure 5. Generated OS codes and simulation models.

process performs a composition of the corresponding code sections as explained in [7].

4.3 Targeting OSs with Simulation Environments as Virtual Processors

As mentioned in section 4.1, the generation of OS simulation models is generating the final 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 of each OS service. In the figure, ovals represent code sections that are processor-independent (denoted with All) or processor-dependent (denoted with 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, then the 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. 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 final 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 $S1$ and $S2$, two C codes $C1$ and $C2$ are used, for the other two OS services $S3$ and $S4$, two assembly codes $asm3$ and $asm4$ are used. In the generated OS simulation model, for $S1$ and $S2$, assuming that the C codes are processor-independent, the same C codes $C1$ and $C2$ are used with delay annotation. For $S3$ and $S4$, their functional simulation models $Func3$ and $Func4$ are used to simulate their functionality with delay annotation. Details of functional simulation models will be given in section 4.4.4.

As shown in the figure, by using the same codes ($C1$ and $C2$ in this example) in both the final OS and the OS simulation model, the code equivalence problem of virtual OS can be mitigated in our method. In the final 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 final OS code.

4.4 Timed OS Simulation Models

4.4.1 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 RPCs (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 SW 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 im-

```
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       > corresponding to
6       LDR     r0,[lr,#-4]       ; Load swi instruction       |  SWI_Enter
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);
```

Figure 6. SWI handler: assembly and C codes.

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

// Counterpart of C code
SWI_Enter() ; delay(24);
__trap_trap(0,id,0);
SWI_Return(); delay(23);
```

Figure 7. A simulation model of SWI handler.

plementation will be given in section 5.

4.4.2 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 [19].

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 handler code of ARM7 processor is shown in Figure 6. The figure shows a SWI handler 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.

```
// fifo_write is an RPC_process
Void OS::fifo_write(int f_id, int data) {

        disable_interrupt(); delay(10);
        // The exec. time of disable_interrupt() is 10 clk cycles.

        if( fifo_full(f_id) == true ) {
                enable_interrupt(); delay(5);
                block(f_id); // task execution is suspended.
                disable_interrupt(); delay(10);
        }

        write(f_id, data);
        enable_interupt(); delay(5); sync_int();
}
```

Figure 8. A simulation model of processor-independent OS code.

To model such exception handlers, our strategy is to model the minimal set of elements to have fast OS simulation. In the case of ARM processor, the minimal set of elements is made of processor mode registers (CPSR and SPSRs) 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 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 (ISRs) 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, a method to model interrupt handling is presented in [20]. 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

```
__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)
{
        delay(34);
        wakeup_event[new_task_id].notify();
        wait(wakeup_event[cur_task_id]);
        delay(3);
}
```

(b) Context switch: simulation model

Figure 9. Context switch: assembly code and simulation model.

the task execution preempted by the interrupt.

4.4.3 Simulation Models of Processor-Independent Codes

In the cases of processor-independent OS codes, to obtain simulation models, we insert delay and sync_int functions into the final OS codes. Figure 8 shows an example of simulation model of processor-independent OS code (a communication service called fifo_write). In the figure, delay and sync_int functions are inserted into the final OS code to simulate the advance of simulation time in the function fifo_write and to simulate the preemption of task execution.

4.4.4 Simulation Models of Processor-Dependent Codes

Examples of processor-dependent codes are boot code, task state code, context switch code, exception handlers, device drivers, etc. For such processor-dependent assembly codes, we use their functional models with performance annotation (using delay functions). Figure 9 shows an example of assembly code and a simulation model of context switch code of ARM7 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, 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

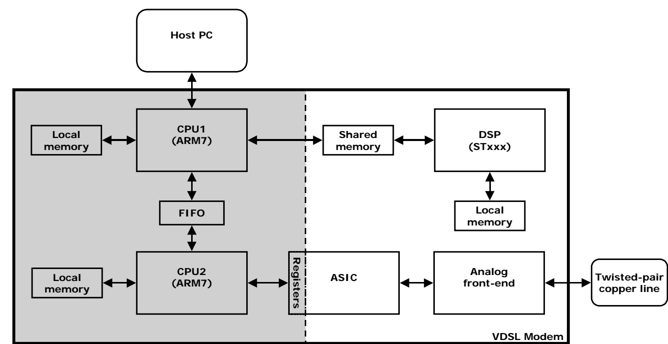

Figure 10. VDSL modem application.

not simulated. One can advocate that for this part we still have a code equivalence problem. This is true. However, since the assembly code occupies generally less than 5% of the total OS code, the code equivalence problem is significantly mitigated.

4.5 Timing Calculation and Application to Configurable OSs

To calculate the execution delay values used in delay functions, we can use conventional estimation methods of SW execution time [17][21][22]. 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 OSs [11], the required steps are as follows.

- Insert delay and sync_int functions into the existing codes of OS.
- Prepare the simulation models of processor-dependent codes.
- Apply the automatic OS generation/configuration flow (in [7]) 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 10. The VDSL modem uses Discrete Multi-Tone (DMT) modulation.[3] 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 10, inside the shaded region. The

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

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 on two ARM7 processors (CPU1 and CPU2 in Figure 10). CPU1 runs three concurrent SW tasks and CPU2 runs six concurrent SW tasks.

To generate OSs, we use our OS generation tool [7]. As a simulation environment, in our experiments, we use SystemC [23]. 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 OSs as shown in Figure 2. We run two types of simulation: one using two ISSs (one for each ARM7 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 ISSs. When the number of ISSs is larger than in our case (two ISSs in our case), this speedup will be even larger due to the synchronization overhead between multiple ISSs. 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 final 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.

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] *AMBA Specification*, ARM Ltd. Hall.

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

[4] J. A. J. Leijten et al., "PROPHID: A Heterogeneous Multi-Processor Architecture for Multimedia", *Proc. Int'l Conference on Computer Design*, 1997.

[5] P. Guerrier and A. Greiner, "A Generic Architecture for On-Chip Packet-Switched Interconnections", *Proc. Design Automation and Test in Europe*, 2000.

[6] W. J. Dally and B. Towles, "Route Packets, Not Wires: On-Chip Interconnection Networks", *Proc. Design Automation Conf.*, 2001.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

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

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

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

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

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

[18] 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.

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

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

[21] 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.

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

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