Automatic HW/SW interface modeling for scratch-pad & memory mapped HW components in native source-code co-simulation

Héctor Posadas[1] and Eugenio Villar[1]

[1] University of Cantabria
ETSIIT, Av. Los Castros s/n,39005 Santander, Spain
{posadash, villar}@teisa.unican.es

Abstract. Native execution of instrumented code is commonly used for early, high-level SW simulations. SW code developed for a target platform is executed in a host computer for fast functional verification and performance estimations. However, as the native platform is different than the target platform, directly writing the peripheral registers or handling scratch pad memories makes the native execution to crash. Previous works require manual recoding to solve this problem. This paper presents a library that automatically solves the problem of simulating directly, fixed memory accesses. HW accesses are detected at run-time in the native execution and redirected to a target platform model. Thus, native HW/SW co-simulation is performed without any recoding effort. Both peripherals only requiring data transfers and peripherals also requiring communication event delivery are automatically managed.

Keywords: High-level modeling, native co-simulation, HW/SW interface, memory access, scratch-pad modeling.

1 Introduction[1]

The constant increase of embedded system complexity is making early, high-level system co-simulations more and more important. Efficient design flows for HW/SW systems requires virtual platforms where the SW code can be developed meanwhile the HW platform is being optimized. Virtual platforms allow designers to verify the SW code functionality. Furthermore, performance information can be obtained to explore the best system architecture, resource mapping or platform configuration.
To obtain early and fast virtual platform models, native execution of the SW code are used. In this simulation technology the SW code is annotated with time information and executed in the host computer together with a TLM model of the target platform. Native execution is much faster than other SW modeling techniques, so it is really useful at the first design steps. Design space exploration, resource allocation and

[1] This work was supported by the Spanish MICyT and EC through MULTICUBE FP7-216693 and the TEC2008-04107 projects.

platform requirement dimensioning can be efficiently performed. The target platform model provides the functionality associated to the HW peripherals to allow correct system execution. The platform model also contains timing information of the HW components. Thus, not only the functional behavior, but also the performance effects can be considered in the system co-simulation.

One of the main problems in native co-simulation is the modeling of HW/SW communication. When directly executing the SW code in the host computer, HW transfers are delivered to the host peripherals, not the target platform model. Thus, the native execution crashes. Previous works require manual recoding the SW to solve the problem. When the HW accesses are performed through system calls, it is feasible. However, when HW is accessed directly reading and writing bus addresses the solution is not valid. In target platforms without MMU, HW accesses are performed through pointer accesses.

HW accesses through pointers are used for HW peripherals, shared memory regions and scratch-pads. When the scratch-pad is controlled by the user, not by the compiler, pointers are set in the code to refer to the scratch pad memory area. These pointers are scattered along the code. In that case, manual recoding is really inefficient, time consuming and error prone. This recoding is even unfeasible if pointer addresses are not fixed but resolved at runtime. Thus, at compilation time it is unknown if the pointer accesses a peripheral or not.

The solution is also valid for Software in the Loop (SiL) simulations. The usual approach for enclosing the interaction between the system and the physical environment is the Hardware in the Loop (HiL) test. Unfortunately, most of the hardware components needed for the test process are available quite late.

To solve that, SiL test has been proposed. Instead of the usage of physical interfaces, software interfaces provided by the operating system are used to connect the SW and the environment model. However, when accessing peripherals through pointers, it is required a method to detect and handle these accesses as if they was performed as system calls. Summarizing, the problem is mainly the same than for virtual platforms.

The only generic way to easily solve this problem is modifying the way the native execution is performed, not the code itself. It is required a method to automatically redirect the HW accesses to the platform model instead to the host peripherals. To redirect the accesses it is required to handle both data value transfers and communication events or only data (Fig 1). This depends on the type of HW component:

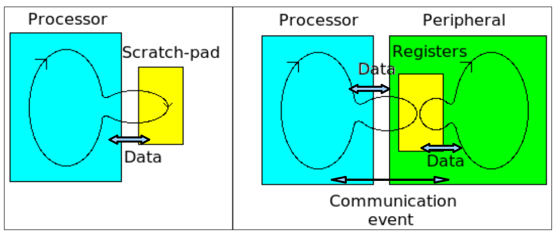

Fig. 1. *Types of Processor-HW communication*

- When accessing HW components like an scratch-pad or a shared memory, only data value management is required. Scratch-pad memories do not perform any functional operation when the processor accesses the data. Thus, in a native co-simulation it is not required deliver a communication event.
- When accessing a HW peripheral, a read or write access usually implies a peripheral action. For example, when some data is delivered to a co-processor, the co-processor must start computing. Thus, these communications does not only requires transferring the values between processor and peripherals; the event must be delivered to the peripheral. Only that way the peripheral model starts performing the adequate actions.

Applying this behavior to native co-simulations is a really difficult task. HW accesses modeled with direct pointer operations do not provoke any event. In the host computer, this is just a read or write operation in a variable.
This paper proposes an automatic technique to dynamically detect data transfers and deliver communication events in native co-simulations. HW accesses from the SW code are detected and redirected at run-time to access a virtual platform model. The technique has been applied to SystemC simulations. As a result, some SW codes developed for target platforms have been simulated in a native co-simulation, without any recoding effort. Data values and communication events are handled independently. This separation allows minimizing the simulation overhead, as not all HW peripherals require both solutions.
Once presented the state of the art, data value management is presented in section 3. Detection and delivering of communication events is described in section 4. The application of these techniques in a native co-simulation environment is presented in section 5.

2 Related work

Integrating models of the HW processors in the system simulation allows obtaining very accurate performance estimations. Commercial tools have solved this challenge [1][2] using ISS that can be connected with a SystemC platform. However, integration of processor models produces a large overload when modeling SW-centric systems. The speed is deeply decreased when running binary code over an ISS with respect to executing directly the source code in the simulation. As a consequence, several approaches have been proposed to improve the results provided by commercial tools. In [3] a SystemC infrastructure developed to model architectures with multiple ARM cores is presented. This approach provides a wide set of tools that allow designers to efficiently design SW applications (OS ports, compilers, etc.). However, it cannot be used to evaluate platforms not based on ARM processors.
In [4] a generic design environment for multiprocessor system modeling is proposed. The environment enables transparent integration of instruction-set simulators and prototyping boards. GDB’s remote debugging features are used to include ISSs in the co-simulation environment.

Another improvement proposed is the modification of the OS running over the ISS. As the OS is in fact the interface between SW applications and the rest of the system, it can be used to save simulation time. In [5], a technique based on virtual synchronization is presented to faster execute several SW tasks in the ISS. Only application tasks run over the ISS. The OS is modeled in the co-simulation backplane. However, although these simulations have improved the simulation speed with respect to commercial tools, the use of an ISS still implies a large overload. Thus, to obtain really fast simulations, the best option is to integrate SW source code directly in the system simulation. ISS accuracy cannot be obtained, but it can be given up in exchange of speed up when dimensioning the system at the first steps of development. Several estimation and annotation techniques have been developed to model SW at in native co-simulation [6-10]. Even commercial tools have been developed to automatically estimate and annotate the SW code [11]. However, SW cannot be adequately modeled only using these techniques. SW requires an OS to execute. When several SW tasks run in the same processor, they cannot run at the same time. They have to be scheduled adequately. Furthermore, SW/SW communication commonly uses mechanisms not included in SystemC. Thus, OS models are required in the simulation environment to model SW execution.

Several works on OS modeling for SW native simulations from abstract OS [12-14] to real OS [15-16] have been proposed. These works also dedicate a large effort in accurately integrating time annotation and OS modeling with HW/SW communication, especially for HW interrupt management. In fact, communication in native simulations has been specifically considered in some works [16-18]. However all this works use function calls to perform communications. Accesses through pointers are not solved in any of them. In case the code contains this kind of accesses manual recoding is unavoidable.

To overcome this limitation, in this paper, an automatic way to perform communication between native simulation and virtual platform models is presented. The technique eliminates the need of SW recoding.

3 Memory remapping of HW data values

In native co-simulations the operating system prevents the application code to access specific HW addresses. When the SW code tries to access a fixed HW address, the memory management unit (MMU) detects a failure as there is no a physical address associated to the required virtual address. As a consequence, the memory management system provokes a segmentation fault. The way to solve the problem is to force the operating system to create a page of virtual memory at the desired memory address. Thus, when the SW under simulation wants to read or write the HW values, values are correctly stored in the host memory.

To force the native operating system to create this memory page, the standard POSIX "mmap" function can be used. The "mmap( )" function shall establish a mapping between a process' address space and a file, shared memory object, or typed memory object. The format of the call is as follows:

"pa=mmap(addr, len, prot, flags, fildes, off);"

The mmap( ) function shall establish a mapping between the address space of the process at an address “pa” for “len” bytes to the memory object represented by the file descriptor “fildes” at offset “off”. The value of pa is an implementation-defined function of the parameter “addr” and the values of flags. A successful mmap( ) call shall return “pa” as its result. To indicate how the system obtains “pa” from “addr”, the parameter “flags” is used (Table 1). Parameter flags provide information about the handling of the mapped data. The value of “flags” is the bitwise-inclusive OR of these options.

Table 1. Possible “flag” and “prot” values for mmap function.

Symbolic Constant	Description	Symbolic Constant	Description
MAP_SHARED	Changes are shared.	PROT_READ	Data can be read.
MAP_PRIVATE	Changes are private.	PROT_WRITE	Data can be written.
MAP_FIXED	Interpret addr exactly.	PROT_EXEC	Data can be executed.
		PROT_NONE	Data cannot be accessed.

To ensure that the memory page created will provide support to the HW addresses required by the SW code under simulation, the option MAP_FIXED must be selected. The parameter “prot” determines whether read, write, execute, or some combination of accesses are permitted to the data being mapped. The “prot” shall be either PROT_NONE or the bitwise-inclusive OR of one or more of the other flags in the following table. For modeling HW memory addresses, PROT_READ and PROT_WRITE flags must be activated.

To apply this solution to a co-simulation infrastructure, it is required to call this function when the platform model is being created. When a peripheral is instantiated, the associated memory address in the target platform is decided. Then the “mmap” function must be called, for the specified address and the indicated memory length. The required code can be shown in figure 2. In that code, a file is created to store the information of the associated memory. It is important to note that the maximum size of the mapped memory is equivalent to the size of the associated file. As a consequence, if an empty file is used, no values can be read or written. The solution applied is to assign a size of “len” to the file before calling “mmap”. To do so, the standard POSIX function “ftruncate” is used.

If the initial address does not correspond with the beginning of a memory page, special management is required. Memory pages always start in an aligned position. Thus the memory activated will start at the corresponding aligned address and will cover “len” bytes. To adjust the addresses, there are two possibilities. First the “offset” parameter can be used to indicate where exactly the mapped memory area must start. The second solution is to increase “len” with the offset of “addr”. In the proposed code (fig 2) the second solution has been used. Furthermore, for debugging purposes is interesting to note that the values stored in the scratch-pad model can be shown by reading the associated file.

```
void initialize_periph (void *addr, int len){
 fd = open("tmp.txt",O_CREAT|O_RDWR,0x01b6);
 ftruncate(fd,get_page_size());

  len += addr - page_aligned(addr);
  mmap(addr,len,PROT_READ|PROT_WRITE,MAP_FIXED|MAP_SHARED,fd,0);
}
```

Fig. 2. *Code for mapping the HW model memory*

The solution is really effective when modeling scratch-pad memories in native co-simulations. It automatically allows executing SW code using fixed HW addresses with a negligible simulation overhead. As no cache misses or any other event is provoked internally by scratch-pad memories, only the ability of reading and writing values at these addresses is required. More specific details of internal scratch-pad operation are not handled at high level.

4 Capturing communication events

HW peripherals as co-processors require receiving information about the communication events. Peripherals are not designed to make polling of any variable. They react to read or write accesses from the system processors. High-level models of the HW peripherals used in native co-simulations emulate their operation mechanisms in the same way. When applying the technique presented in the previous section, the storage is done but the peripheral does not receive any event informing that a read/write operation has been performed in their registers.
The only way to produce the event is to not apply the solution of the previous section at the beginning of the simulation and let the simulation crash. When the simulation is going to crash due to the segmentation fault, the error can be captured, solved and then the simulation can continue. The result of that process is that the HW access is detected and the event can be sent to the peripheral model. When the SW tries to access an invalid memory address, the native operating system raises a SIGSEGV signal. This signal can be captured with an appropriate signal handler. This prevents the program to terminate. However, the HW access cannot be performed at the signal handler. Neither the access type (read/write) nor the value are known at the handler.
To obtain the data, the memory remapping technique transfer presented in the previous section is used. At the signal handler the memory mapping is activated and the code returns to repeat the pointer access. To perform a correct access, in reading accesses, the read transfer to the virtual platform is done first, updating the adequate memory address. Thus, when retrying the pointer read, the value obtained is correct. For writing accesses, the pointer access is performed first, and after that the value written is sent to the virtual platform.
Performing an “mmap” allows retrying the instruction, but once an access has been performed, the memory page is active and further accesses are not detected. To solve that, the memory page must be unmapped. However, when the code returns to the failed instruction and it continues normally, that is, without unmaping the page. A

possible solution is to create a parallel thread that wait a certain time and then unmaps the page. However, this is a really unsafe solution. There is no guarantee that there will be no more accesses before the unmap step, and even there is no guarantee that the unmap is done once the application SW code continue the native simulation.

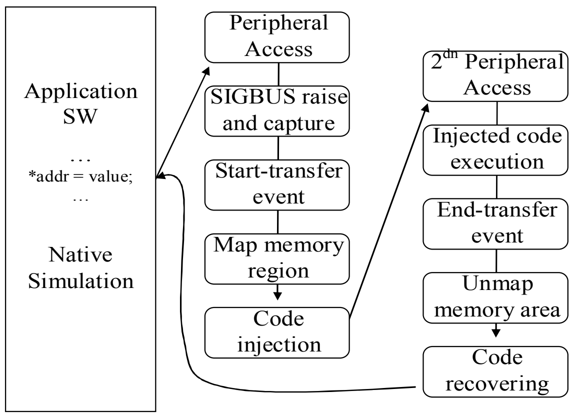

Fig. 3. *Process for complete handling of HW accesses directly using pointers*

To unmap the memory page properly, the SW code itself must do it. Just after the memory access is performed, the page must be unmapped. To do that, the original SW code must be modified. The solution applied is to dynamically inject code after the load/store assembler instruction that provoked the error. This injected code disables the memory page, re-establish original SW code and continues the execution. As a consequence, the HW access is performed, the peripheral model is informed and the simulation status returns to the correct point to detect new accesses. Although the memory page is unmapped the data stored are not lost. The values are saved in the file associated to the memory page. The entire process is summarized in figure 3.

Detecting if a pointer access is a reading or a writing one is also complex. A possible solution is to disassemble the binary code of the instruction provoking the error, but this solution is non portable. Furthermore, in x86 processors both reading and writing accesses are performed with “mov” instructions, so it is not easy to distinguish both.

The portable solution is to force the system to raise different signals for read and write accesses. When executing an I/O pointer access, a SIGSEGV signal is obtained if the memory address has not been mapped. If the address has been mapped but the associated file has 0 size, a SIGBUS signal is raised. Thus at initialization the address is only activated for reading accesses with an empty file. Thus, a SIGSEGV raises at writing accesses (there is no writing permission) and a SIGBUS raises at reading accesses (there is no area in the associated file).

4.1 Capturing signals

When the peripheral address is accessed, the memory manager of the native operating system raises a SIGBUS or a SIGSEGV signal. These signals can be captured using an interrupt handler that can be loaded using the standard POSIX “signal” function.

```
void signal_handler(int sig, siginfo_t* info, void* data){

    bus_address = (int)info->si_addr;
    if(!is_HW_addr(bus_address)) raise(SIGINT);

    unmmap(bus_address);
    file = get_no_empty_file(bus_address);
   mmap(bus_address, LEN, PROT_READ|PROT_WRITE, MAP_FIXED, file, 0);

   if(is_read = (sig == SIGSEGV))
        * bus_address=bus_read(bus_address);
    Inject_code(data);
}
```

Fig. 4. Signal handler for SIGBUS and SIGSEGV signals

The handler (fig 4) obtains the address provoking the error and checks that it is a valid I/O access. Using the data address the required memory region can be mapped to allow a retry. The active memory mapping with read-only access and an empty file is replaced by a read/write access with a valid file. Once the memory is mapped, the code injection must be performed.

4.2 Code injection

To guarantee the memory region is unmapped properly, a new code must be injected after the peripheral access. To inject the code (fig 5), the memory region where the code will be placed is declared a read/write region, using the "mprotect" function call. Then the original code is saved in a buffer and the new code is injected.

```
void Inject_code(ucontext_t *ucp){
     struct sigcontext *sc;
     sc = (&(ucp->uc_mcontext))->gregs;
     as_addr = sc->eip + instruct_size(sc->eip);

     mprotect( page, getpagesize(), PROT_READ|PROT_WRITE|PROT_EXEC );

     memcpy(backup, as_addr, injectSize);
     memcpy(as_addr, &injectStart,  injectSize);
}
```

Fig. 5. *Code in charge of performing the code injection*

To make the solution portable for C-based simulation environments, the code to be injected is also written in C, avoiding specific assembler code. Two "asm volatile" marks are added to the C code to know where it starts and ends. The code injected has to be small (fig 6). The injected code is composed just by two function calls: one to get the current context and one to perform the writing access and the system recovering.

```
int (*getContext)(ucontext_t *ucp)=&getcontext;
void (*recoverFunction)() = &recoverFunction;

asm volatile( "injectionStart:" );

(*callGetContext)( &uc_auxiliar_ucp );
(*callreturningFunction)( );

asm volatile( "injectionEnd:" );
```

Fig. 6. Code to be injected

4.3 System recovering

The recovering function (fig 7) starts performing a writing access in the HW platform model if required. Then the function unmaps the memory region using the “unmmap” function, and maps the address in read-only mode with an empty file. The original code is recovered using a “memcpy” function call, and the processor status is restored, continuing with the normal execution. The restoring of the processor status is preformed using the “setcontext” function from the "asm/sigcontext.h" library. When restoring the instruction pointer, the execution jumps to the initial code, after the pointer access.

```
void recoverFunction(){

    if(!is_read) bus_write(bus_address,*bus_address);

    unmmap(bus_address);
    file = get_empty_file(bus_address);
    mmap(bus_address, LEN, PROT_READ, MAP_FIXED, file, 0);

    memcpy(as_addr, backup, injectSize);
    setcontext(&uc_auxiliar_ucp);
}
```

Fig. 7. *Function that recovers the initial status to continue the simulation*

5 Application into a native co-simulation tool

The solutions presented above have been applied to a state-of-the-art native co-simulation infrastructure to check their validity. SCoPE tool has been selected for this purpose. The selected infrastructure provides facilities to generate HW platform models (fig 8). Performance estimations of the SW code and the entire system can be obtained. The infrastructure also provides a complete RTOS model. This RTOS model allows directly executing SW code developed for a target platform. Recoding of the system calls is not required to perform the SW native simulation.
When applying the solutions proposed in this paper with the SCoPE features, the target SW code can be automatically simulated. The access to both SW (RTOS) and HW resources (peripherals) is dynamically handled by the simulation engine. To

extend SCoPE with the proposed solutions for HW communication modeling, it is required to load the bus error signal handler and to create a module that generates the required bus accesses when the communication events are delivered. To do so, the functions "bus_model->transport" can be used to send the transfers through the bus of the platform model. No other modifications have been required. This means that the proposed solutions can be also easily applied to any other native co-simulation infrastructure.

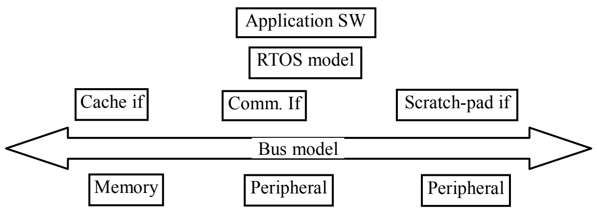

Fig. 8. *Platform model built using SCoPE tool*

To show the usefulness of the proposed solutions, an example of a GSM system has been proposed. The GSM system is composed of the coder and the decoder. Each part contains several tasks that can be executed concurrently (fig 9). Input and output values are sent and received using specific I/O HW components. The code has been prepared for an ARM based platform running uclinux. Using SCoPE and the proposed extensions, the target code was automatically integrated in the native co-simulation without any additional effort. However, this automation increases simulation time. To obtain the simulation overhead three different simulations have been performed:

- A coder without I/O HW accesses (all SW)
- A coder with I/O HW accessed by function calls
- A coder with I/O HW accessed through pointers

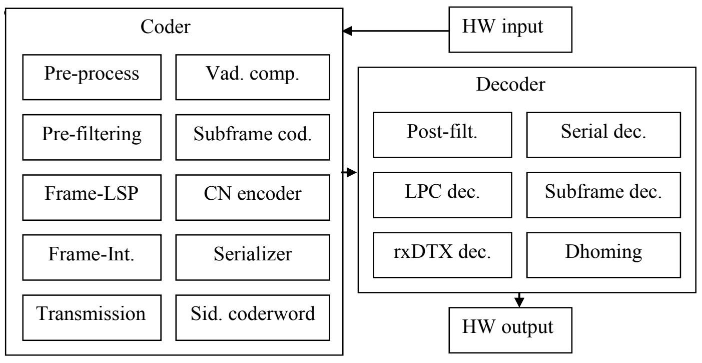

Fig. 9. *Coder – decoder task graph*

The result obtained is that the pointer access technique proposed in this paper duplicates the simulation time cost of I/O accesses w.r.t HW communication techniques based on function calls (Table 2).

Table 2. Simulation time for the GSM coder (285 frames)

All SW	Function accesses	Pointer accesses
11.3 sec	12.1 sec	13.0 sec

The code has been used to explore different platform architectures in order to select the best one. Mono-processor architecture, multiprocessor symmetric and heterogeneous architectures, and network-based architectures has been explored. The obtained results are shown in the following table (Table 3).

Table 3. *GSM Performance information: time, power and processor utilization*

Estimated time (sec)	MonoP	SMP	HMP	Net
Proc 1	60.2 s	32.2	55.08s	55.15s
Proc 2	-	28s	5.2s	5.5s
Total	63.6s	34.7s	59.8s	60s
Estimated Energy (mJ)	MonoP	SMP	HMP	Net
Proc 1	149	80	142	136
Proc 2	-	69	77.4	13
Total	153.4	218.1	219.4	218.2
Processor utilization	MonoP	SMP	HMP	Net
Proc 1	95%	92%	92%	93,00%
Proc 2	-	82%	9%	8%

Although the estimation technique is not part of this work, it is interesting to note that source-level estimation techniques have demonstrated to obtain errors lower than the 20% in timing and processor power consumption. This is considered a sufficient accuracy for system dimensioning and analysis at first steps of development.

6 Conclusions

Automatic integration of SW code developed to target platforms can be integrated in native co-simulations. To do so, direct I/O communications from the SW code must be intercepted and redirected to virtual platform models instead of the native host peripherals.
I/O communications has been divided in two groups for native modeling: communications only requiring data load and store and communications required generating events. Modeling accesses to HW components only requiring data storage management can be easily performed by using the memory mapping facilities of the native operating system. Modeling access requiring event generation needs handling the memory faults and injecting additional code in the original execution.
Both techniques can be done using standard functions. The use of functions contained in the POSIX standard has been demonstrated. This characteristic makes the solution

portable to a wide range of host computers, as Linux or Unix. The solution requires only setting a signal handler so they can be easily applied to any simulation engine.

References

1. Coware Platform Architect, www.coware.com
2. ARM Realview Development Suite, www.arm.com
3. L. Benini, A. Bogliolo, F. Menichelli: "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC", Journal of VLSI Signal Processing, Springer, 2005
4. L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi and M. Ponzino: "SystemC cosimulation and emulation of multiprocessor SoC design", IEEE Computer, April, 2003.
5. Y. Yi, D. Kim and S. Ha: "Fast and time-accurate cosimulation with OS scheduler modeling", Design Automation of Embedded Systems, N.8, Springer, 2003.
6. C. Kirchsteiger, H. Schweitzer, R. Weiss and M. Pistauer: "A Software Performance Simulation Methodology for Rapid System Architecture Exploration", ICECS, 2008
7. J. Schnerr, O. Bringmann, A. Viehl and W. Rosenstiel: "High-Performance Timing Simulation of Embedded Software", Proc. Of DAC, 2008.
8. C. Brandolese, W. Fornaciari, F. Salice, D. Sciuto, "Source-level execution time estimation of C programs", Proc. of CoDes, 2001.
9. T. Kempf, K. Karur, S, Wallentowitz, H. Meyr, "A SW Perfornance Estimation Framework for Early SL Design using Fine-Grained Instrumentation", Prof. of DATE, 2006
10. Y. Hwang, S. Abdi, D. Gajski: "Cycle approximate Retargetable Performance Estimation at the Transaction Level", Proc. of DATE, 2008
11. InterDesign Technologies, FastVeri http://www.interdesigntech.co.jp/english/
12. A. Gerstlauer, H. Yu, D.D. Gajski, "RTOS Modeling for System Level Design", Proc. of DATE, IEEE, 2003.
13. Z. He, A. Mok and C. Peng: "Timed RTOS modeling for embedded System Design", Proc. of RTAS, IEEE, 2005.
14. S. Yoo, G. Nicolescu, LG. Gauthier and A.A. Jerraya: "Automatic generation of fast timed simulation models for operating systems in SoC design", Proc. of DATE, 2002.
15. Hassan M.A., Yoshinori S., K. Takeuchi, Y. & Imai, M. "RTK-Spec TRON: A Simulation Model of an ITRON Based RTOS Kernel in SystemC". Proc of DATE, 2005.
16. J. Castillo, V. Fernández, H. Posadas, D. Quijano, E. Villar: "SystemC Platform Modeling for Behavioral Simulation and Performance Estimation of Embedded Systems", in the book "Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation". IGI international ed.
17. A. Wieferink, R. Leupers, G. Ascheid, H. Meyer, T. Michiels, A. Nohl and T. Kogel. Retargetable generation of TLM bus interfaces for MPSoC platforms. CODES+ISSS'05.
18. P. Gerin, X. Guérin, F. Pétrot: "Efficient Implementation of Native Software Simulation for MPSoC". Proc. of DATE, 2008.