Binary Replacement Technique for Application Programming Interface Level Simulation

Junghee Lee[1], Denis Paterson[2], Scott O'Neill[2], Hongchul Kim[3], Sangjun Nam[3], Jongman Kim[1]

1: Georgia Institute of Technology, 777 Atlantic Drive NW, Atlanta, GA 30332, USA
2: Synopsys, The Alba Centre, The Alba Campus, Livingston, West Lothian, Scotland, UK
3: Samsung Electronics, Telecommunication R&D Center, Yeongtong-gu, Suwon-si, Gyeonggi-do, Korea

Abstract: Design of complex embedded software requires ingenious solutions to many architectural problems. One such solution that would be a crucial catalyst in designing scalable and customized embedded software, is developed by API (Application Programming Interface) level simulator. The use of API level simulator has been gaining wide acceptance due to its design and verification efficiency by enabling parallel development in multiple software layers. However, there are two major bottlenecks in realizing practical systems: source code modification and recompilation of the target software. The paper proposes a novel simulation technique to resolve these two critical issues. The proposed technique makes it possible to replace *any* part of the target binary without modifying its source code and recompiling it.

Keywords: API level simulation, embedded software, instruction set simulator

1. Introduction

Developing software is becoming a major bottleneck of developing embedded systems as its complexity grows. Simulators can shorten time-to-market by parallelizing development of software into independent development of layers. The register accurate level simulator behaves the same as the target hardware. The software running on the simulator is compiled by the target compiler. Here, *target* means an embedded system to be implemented, while *host* means a system, usually a personal computer or a workstation, where a compiler, debugger, and simulator are executed [8]. To simulate not the whole software but only the upper-layer software such as OS (Operating System), middleware, and applications, API (Application Programming Interface) level simulators are used.

API level simulator provides the same set of APIs that behave the same as the underlying layer of the target software although their implementation may be different. The software to be simulated on API level simulators is required to be compiled by a host compiler and liked with the library that is a substitute for the underlying layer. Thus, source code modification and recompilation of the target software are inevitable. Examples of API level simulators include VxSim for simulating VxWorks [5], a simulator included in Nokia S60 platform SDK [6], and Platform Builder for Windows Mobile [7].

There have been techniques that redirect OS or middleware function calls of the target binary to corresponding OS or middleware function calls of the host [11-15]. They don't require source code modification and recompilation. However, they are restricted to the given API layer. They don't allow the developers of the API level simulator to change the interface where the function calls are redirected.

To address those issues, a novel simulation technique is proposed. The main contribution of this paper is to introduce a binary replacement technique. We extend semi-hosting [9] for API level simulation to a modeling concept, *interception*, and introduce a new concept, *invocation*. The proposed technique makes it possible to replace *any* part of the target binary with the host binary without source code modification and recompilation of the target software. Since the proposed technique allows designers to select which part of the target binary should be replaced, the designers can use this technique for developing any abstract level simulators.

In section 2, related works are discussed in detail. Section 3 explains the proposed technique. Preliminary experimental results are provided in section 4 and a conclusion is made in section 5.

2. Related Works

Table 1 compares characteristics of the proposed technique with those of existing simulation techniques that can be used for general API level simulators, not restricted to a certain API level.

To our best knowledge, all the commercial API level simulators including [5-7] have been implemented with the host code execution technique. The host code execution technique requires the

Table 1: Comparison of simulation techniques

	Abstraction level	S/W compile	Source code modification	Target binary execution	Simulation speed
Host code execution	API	Host	Required	None	Fast
Semi-hosting	API	Target	Required	Above API	Middle
Proposed technique	API	Target	No	Above API	Middle
Register accurate	Register	Target	No	Full	Slow

software to be modified and recompiled by the host compiler as mentioned before. Thus, the target binary is not executed at all. Instead, it executes the binary whose source code is modified and compiled by the host compiler. Its simulation speed is fast because it doesn't require interpreting the target binary.

Source code modification makes it difficult to maintain the source code by the *developers* who take charge in implementing the API level simulator. Usually, the API level simulator enforces modifying parts of the target software to accommodate the new implementation for the simulator. It implies that the source code of both the target software and the simulator should be maintained concurrently. Nowadays, maintaining even single source code is not easy due to its huge size. What makes it more difficult is that they should be developed by multiple developers simultaneously.

However, *users* don't need to maintain two different versions of their software because the developers should have taken care of that. Here, users mean those who use the API level simulator to develop their software. The users simply turn on and off a switch like define pragmas, which are provided by the developer wherever they may be needed. However, the users still need to recompile their software, which also causes practical issues regarding different results of compilers and third-party libraries whose source code is not provided.

Ideally, changing compilers must not cause any problem because the software to be simulated is independent of the platform on which it is executed. In reality, differences between the results from different compilers, which are usually related to data structures, sometimes cause problems. Users sometimes assume a certain result from the compiler, which can be incorrect when the compiler optimizes their software aggressively. Although this does not happen frequently, when it does, an in-depth investigation is required to figure out the root cause.

Embedded software in commercial products often employs third-party libraries whose source code is not provided. They cannot be simulated by the host code execution technique because they cannot be recompiled. It limits the coverage of the target software to be developed with the simulator.

The semi-hosting technique of ARM [9] doesn't require recompilation but still needs to modify the source code. An SWI (Software Interrupt) instruction should be inserted into the source code so that the ISS (Instruction Set Simulator) can catch the SWI to redirect to an alternative function that communicates with the debugger. For the upper layer software above the API, the target binary is executed, while for the underlying layer, debugger's code is executed. Its simulation speed is in-between that of the host code execution technique and the register accurate technique. However, our experimental results show that when its abstraction level becomes lower, it becomes as slow as the register accurate technique.

Since the semi-hosting technique is originally not for the API level simulation, there are limitations on developing general API level simulators based on the semi-hosting technique. Most of all, it doesn't provide a way to simulate an incoming event from the underlying layer like a hardware interrupt.

The register accurate technique enables to simulate the whole target binary without source code modification and recompilation. This is because register accurate hardware models are used as substitutes of the real hardware. However, there are also practical issues regarding simulation speed and development time if it is considered to be used for API level simulation.

Recently, some commercial tools [1-4] are reported to achieve very high simulation speeds, even faster than the real target by utilizing the binary translation technique [8]. The biggest obstacle for them to be adopted for API level simulation is that they enforce serialization of development. For example, if one wants to use the register accurate level simulator for developing application software, he should wait for completion of hardware modeling as well as the porting of all the underlying software such as device drivers, OS and middleware which also need to be completed serially.

Emulation [16], virtualization [17], and binary translation technique [18] can also allow for interchangeable execution of binaries that are compiled on different platforms without recompilation. There have been also techniques that redirect OS or middleware calls of the target binary to corresponding OS or middleware calls of the host [11-15]. They are similar to the proposed technique in that source code modification and recompilation are not required. However, *they don't allow for developers to change the interface where the binaries are interchanged*. In contrast, the proposed technique allows for developers to replace *any function* of the target binary to *any function* of the host binary.

3. Binary Replacement Technique

This section explains how to replace a part of the target binary by the host binary. Figure 1 illustrates the key concepts of the proposed technique. While the target binary is being simulated, if calling `FuncA` is detected, it is intercepted and an alternative function `FuncA′` is executed instead of `FuncA`. Since the target binary cannot be executed directly on the host machine, it is simulated on the ISS. In contrast, `FuncA′` can be executed on the host machine because it is compiled by the host compiler. From now on, we call the alternative function like `FuncA′` and auxiliary functions making it possible for the alternative function be executed as a *Host eXtension Module* (HXM).

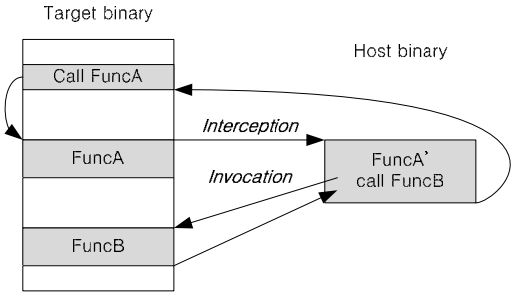


Figure 1: Key concepts of the binary replacement technique

Invocation is also necessary to model incoming events. Invocation makes the ISS branch forcefully to a certain function. This functionality cannot be implemented with the semi-hosting technique [9]. Suppose that a function above API should be called if a key is pressed. Most middleware provides an API for registering an event handler. Once an event ID for the key press event and its event handler are registered, the function would be called if the event occurs. Without invocation, there is no way to simulate such a case. In addition, common libraries in the target software can be reused in an HXM by invoking them. Without invocation, the libraries should be also implemented identically in an HXM.

The ISS should provide interfaces to make it possible for an HXM to access the target memory by the target address. The HXM needs to access the target memory to communicate with the target software.

Table 2 summarizes communication interfaces that are provided by the ISS for an HXM to communicate with the target software running on the ISS. By using these interfaces any part of the target binary can be replaced.

3.1 Procedure Definition

Figure 2 shows the procedure of the interception and the invocation. Each interface in table 2 is explained following the procedure.

During initialization the ISS calls an initialization function in the HXM, which calls `RegisterInterceptionFunc` to register a function to be intercepted. The start address of the function and the pointer of the alternative function in the HXM should be provided. Here, the address is of the simulated target while the pointer is of the host.

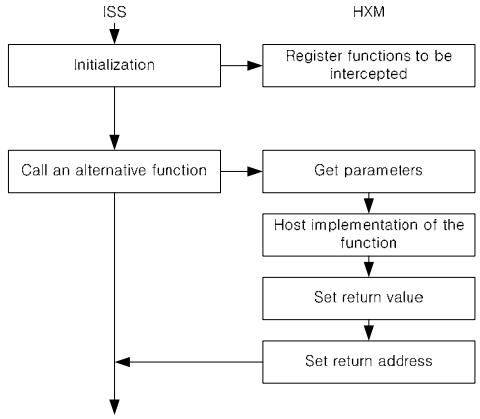
The ISS calls the alternative function when the PC becomes equal to one of the registered start addresses. In order to avoid deteriorating the simulation speed, the ISS doesn't actually monitor the PC every cycle in our implementation. Instead, we exploit the simulation mechanism of the ISS. The ISS [1] employs dynamic compilation-based approach [10]. The technique is that it doesn't emulate each instruction of the target binary but instead translates a block of target instructions into corresponding instructions of the host. In the case of intercepted functions, it generates instructions to call the alternative function instead of calling the corresponding instructions.

The alternative function in the HXM gets parameters according to the calling convention of the target code. For an example of the ARM compiler, first four parameters are stored in `R0`, `R1`, `R2`, and `R3` registers and others are stored in the stack. `GetRegisterValue` and `ReadISSMemory` are used to read registers and contents in the memory. Contents of the stack can also be read by `ReadISSMemory` since the value of the stack pointer can be obtained by `GetRegisterValue`. When calling `GetRegisterValue`, the index of the register to be read and a pointer where the value is to be stored should be provided. `ReadISSMemory` reads data from the simulated memory.

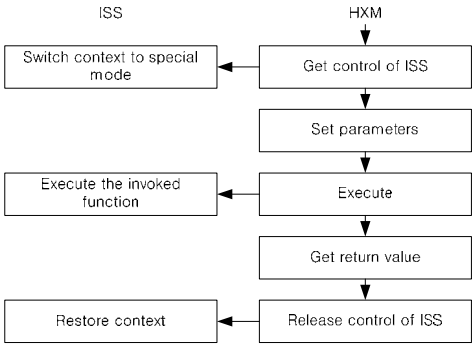
The HXM then executes the host implementation of the function, which the user wishes to run to replace the function of the target binary. The host function may need to access global variables or to de-reference pointers, both of which would be resident in simulated memory. `ReadISSMemory` and `WriteISSMemory` are used for those purposes. `WriteISSMemory` writes data to the simulated memory.

Table 2: Summary of communication interfaces

Category	Name	Parameters	Description
Interception	RegisterInterceptionFunc	Start address of the intercepted function Pointer of the alternative function	Register functions to be intercepted
Invocation	GetControlOfISS	None	Make the ISS switch context to the special mode
	ExecuteFunctionCall	Start address of the invoked function	Execute the invoked function
	ReleaseControlOfISS	None	Make the ISS restore the context
Register access	GetRegisterValue	Index of the register Pointer where the value is to be stored	Read register value
	SetRegisterValue	Index of the register Pointer where the value is stored	Write register value
Memory access	ReadISSMemory	Address of the data to be read Size of the data to be read Pointer where the data is to be stored	Read data from the simulated memory
	WriteISSMemory	Address of the data to be written Size of the data to be written Pointer where the data is stored	Write data to the simulated memory



(a) Procedure of interception



(b) Procedure of invocation

Figure 2 : Procedure of interception and invocation

The return value of the intercepted function is set according to the calling convention by calling `SetRegisterValue` or `WriteISSMemory`, if necessary. `SetRegisterValue` writes a value to a register.

Finally, the value of the PC should be set to the return address. For an example of the ARM, the return address is stored in the link register. Thus, the value of the PC should be set to that of the link register. `GetRegisterValue` and `SetRegisterValue` are used to read the link register and to write to the PC. This allows the ISS to continue executing the target binary from the return address of the intercepted function.

In order to perform invocation, the ISS is required to conserve its context before and after the invocation. For this purpose, the special mode is introduced. Like the IRQ mode of the ARM, the ISS stores and restores all the registers to and from the memory when switching the context. The HXM should first call `GetControlOfISS` to make the ISS switch context to the special mode before invoking a function. Then, the HXM sets parameters according to the calling convention by `SetRegisterValue` and `WriteISSMemory`.

`ExecuteFunctionCall` is called to execute the invoked function on the ISS. The start address of the invoked function should be provided so that the ISS branches there.

Then, we need to determine when the invocation is over. Since the invoked function may call another function, it cannot be determined by simply detecting return instructions. The way we resolve this issues is as follows. When the ISS branches to the invoked function, it sets the return address to an address which is not a valid code address. By detecting a branch to that address, it can determine when the invocation finishes.

After the HXM gets the return value, it calls `ReleaseControlOfISS` to let the ISS restore the context, and continue executing from the point where it was when the invocation began.

3.2 Iterative Interception and Invocation

Interception and invocation need to be managed as a stack respectively to deal with iterative interception and invocation. Suppose that an invoked function `FuncA` calls `FuncB` which is supposed to be intercepted so that its alternative function `FuncB'` is to be executed. If `FuncB'` invokes `FuncC`, `FuncC` should be executed while the invoked function `FuncA` is not yet finished. To deal with this situation, invocation needs to be managed as a stack and so does interception.

However, invocation sometimes needs to be managed as a queue. Suppose that if a user pushes buttons on the simulator's GUI (Graphic User Interface), a certain function in the target binary should be invoked. If the user pushes buttons A and B in quick succession, pushing B before the invoked function for pushing A has completed, invoking the function to handle B needs to wait. If this case were to be implemented as a stack, pushing B would be processed first, which would be incorrect behavior. This case happens only when invocation is triggered by an asynchronous event from outside the simulator. This case is not an issue between the ISS and the HXM but between the HXM and the external source of the event. Thus, the HXM takes charge of implementing a queue for asynchronous events. When an asynchronous event is received, it is pushed onto the queue and processed when the HXM is scheduled by the simulation kernel. When the HXM is scheduled, it pops an event from the queue and invokes the corresponding function. The return value is passed via a callback function.

The intercepted function may need to be invoked again. This case happens typically if the developer wants to add a behavior to the function instead of replacing it. To deal with this case, the ISS needs to

keep track of which functions are being intercepted. If the invoked function is on the list, it should not be intercepted again.

3.3 Virtual Addressing

There are two cases where virtual addressing is engaged. One is demand paging and the other is assignment of independent virtual address spaces to individual processes. They are often used together.

Demand paging is generally used for embedded systems that have a smaller RAM (Random Access Memory) than a NVM (Non-Volatile Memory). If the software attempts to access a region that doesn't reside in RAM but in NVM, a page fault occurs. The page fault handler copies that region from NVM to RAM and changes the mapping information between virtual and physical addresses accordingly. Demand paging can be simulated by the proposed technique as long as all the addresses used for interfaces are virtual and the ISS generates a page fault whenever it occurs even in the special mode.

Assignment of independent virtual address spaces to individual processes imposes a limitation on interception. Only functions with fixed virtual and physical addresses can be intercepted. This is because the function which is actually intercepted depends on which process is scheduled at that time even though the same virtual address is used.

The same limitation is also imposed on invocation by an asynchronous event. Only functions with fixed virtual and physical addresses can be invoked by an asynchronous event. The developer may implement the HXM to check which process is scheduled before invoking a function. However, it is not a safe way because the atomic operation of updating the mapping information between virtual and physical addresses and updating the scheduling information may not be guaranteed by the simulation kernel. The OS running on the ISS can guarantee the atomic operation but the simulation kernel cannot. The simulation kernel may switch the scheduled model from the ISS to another model such as a HXM, while the ISS is simulating somewhere in the middle of the atomic operation. On the other hand, there is no problem with normal invocation. Since normal invocation is triggered from within intercepted functions, the developer can know which process is being scheduled at that time.

4. Hiding Housekeeping Tasks

Housekeeping tasks mean auxiliary behaviors to make interception and invocation work such as registering functions to be intercepted, getting and setting parameters and return values, and getting and releasing control of the ISS. Communication interfaces are mostly called by housekeeping tasks.

This section describes *HXCreator*, which is used to hide housekeeping tasks from developers so that they can develop HXMs as if they developed a part of the target binary, without requiring an in-depth knowledge of the simulation technique and the calling convention.

Figure 3 shows an example work flow of interception. HXCreator reads the map file and generates the housekeeping tasks, skeleton code and compilation environment such as `makefile`. Grey boxes in the figure are to be provided by the developer and the user. The developer should provide the prototype of a function to be intercepted in the map file. Once HXCreator generates the skeleton code, the developer implements its body. The symbol table should be provided by the user at run-time.

The map file should contain the name, return type, calling convention, number of parameters and type of each parameter of the function to be intercepted. The prototype of the function in C or C++ contains all the necessary information. In this example, HXCreator generates `WrapFuncA` which takes charge of housekeeping tasks for `FuncA`. HXCreator also makes `WrapFuncA` registered in the init function so that `WrapFuncA` is called when `FuncA` is intercepted. The address of `FuncA` is read from the symbol table at run-time. `WrapFuncA` is generated to read parameters according to the calling convention and their type, to call the alternative `FuncA` in HXM whose body is implemented by the developer, and finally to set the return value.

Similarly, the housekeeping tasks for invocation can be hidden. The developer provides the prototype of a function to be invoked in the map file, too. The map file should be able to distinguish which function is to be intercepted or invoked. Suppose that `FuncB` is to be invoked. The developer just calls `FuncB` as if he calls `FuncB` in the target binary directly. `FuncB`

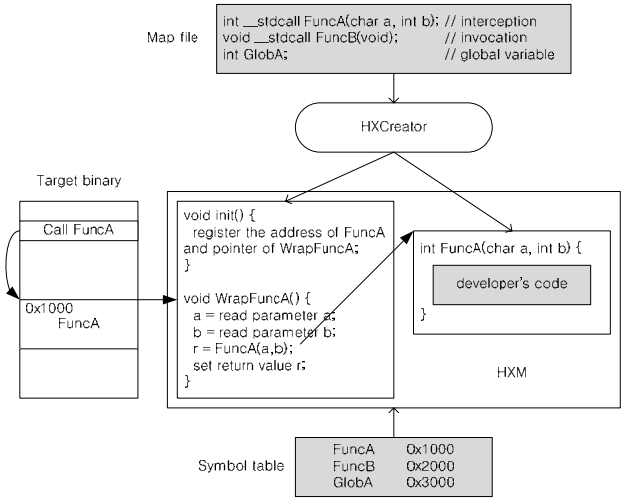


Figure 3: Generating housekeeping tasks for interception

was actually generated by HXCreator as a member of HXM and takes charge of housekeeping tasks for invoking `FuncB` such as getting control of the ISS, setting parameters, requesting execution, getting return value and releasing control of the ISS.

As for global variables and pointers, wrapper classes are introduced. Instead of accessing them by their address, the developer can access them by calling member methods or using overloaded operators of their wrapper class. HXCreator generates housekeeping tasks and initialing code for the wrapper classes.

For example, a global variable `GlobA` is to be accessed. The developer inserts `GlobA` into the map file with its type. It also looks like a variable declaration in C or C++. HXCreator generates declaration of `GlobA` as an instance of the wrapper class. HXCreator also generates its initialing code that reads the address of variable `GlobA` from the symbol table and stores the address in the class `GlobA`. If the developer wants to write to the variable `GlobA`, he can do so by simply using assignment since the assignment operator is overloaded. To read, he may use implicit or explicit type casting operators.

5. Experiment

The proposed technique was successfully incorporated with the Innovator [1]. To compare all the simulation techniques mentioned in section 2, we had to use a very simple test case. This was because we could not implement complicated simulation models using the semi-hosting technique. The test case was to display messages repeatedly on the debugging console via UART. Table 3 shows the simulation results. All the test cases are implemented with the Innovator [1].

When the test case is implemented with the host code execution technique, 4 loc (Lines of Code) needed to be modified. The modification is mostly removal of hardware initialization such as UART and an interrupt controller. The value of 4 loc only counts the removal of lines of code which call the initialization functions. If the removal of their body is also counted as modification, the amount of the modification is 566 loc. As mentioned in section 2, the target binary is not executed at all. Instead, the binary executed is that whose source code had been modified and recompiled by the host compiler. The simulation speed is the fastest among the simulation techniques.

To implement with semi-hosting technique, 3 loc of the source code are modified. The target binary has also been changed to be linked with the libraries that support the semi-hosting technique. Only the target binary above the API, which displays messages in this test case, is executed. The ratio is measured by loc. Its simulation speed is expected to be in-between that of the host code execution technique and the register accurate technique. In this test case, its simulation speed is as slow as the register accurate technique. Since this test case is quite simple, the overhead of communicating with the debugger is significant.

Table 3: Experimental results

	Source code modification	Target binary execution	Simulation speed
Host code execution	4 loc (566 loc)	0 %	2 sec
Semi-hosting	3 loc	50.6 %	39 sec
Proposed technique	No	50.6 %	28 sec
Register accurate	No	100 %	39 sec

With the proposed technique, the source code doesn't need to be modified. However, it still requires efforts to develop HXMs. The difference is that the source code of the HXM can be maintained completely separately from that of the target software. The ratio of the executed target binary is the same with that of the semi-hosting technique. Its simulation speed is faster than that of the register accurate technique but slower than that of the host code execution technique. If the abstraction level of API goes up higher, the gap between the simulation speed from that of the register accurate technique would become larger.

With the register accurate technique, the target binary can be used without any modification and the whole target binary was executed. However, 49.4% of the target binary is below the API and out of our interest. Its simulation speed is slowest among the simulation techniques.

6. Conclusions

In this paper, a novel simulation technique is proposed to address practical issues of conventional API level simulators. Interception and invocation are introduced and their procedure and interfaces are defined. The proposed technique makes it possible to replace any part of the target binary by the host binary without source code modification and recompilation of the target software. The proposed technique is successfully incorporated with the Innovator [1]. As a result, we are able to remove obstacles for the simulators to be adopted by developers and users. With the proposed technique, developers don't need to manage two versions of the source code concurrently. Users don't need to make efforts to port their software with a different compiler and become able to simulate their software with third-party libraries whose source code has not been provided.

7. References

[1] Innovator. Available: http://www.synopsys.com

[2] System Generator. Available: http://www.arm.com

[3] P.Magnusson et al., "Simics: A full system simulation platform," *IEEE Transactions on Computers*, Volume 35, Issue 2, pp.50-58, February, 2002

[4] CoMET. Avaiable: http://www.vastsystems.com

[5] VxWorks. Available: http://www.windriver.com

[6] S60 Platform. Available: http://www.forum.nokia.com/main/platforms/s60/

[7] Windows CE, Available: http://www.microsoft.com

[8] J. Zhu and D. Gajski, “An ultra-fast instruction set simulator,” *IEEE Transactions on VLSI*, Volume 10, Issue 3, pp. 363-373, June, 2002

[9] Angel Debug Protocol Messages. Available: http://www.arm.com

[10] R. Cmelik and D. Keppel, “Shade: A fast instruction-set simulator for execution profiling,” in *Proc. of ACM SIGMETRICS Conference on Measurement Modeling Computer Systems*, 1994, pp.128-137

[11] M. Krause et al., “Combination of instruction set simulation and abstract RTOS model execution for fast and accurate target software evaluation,” in *Proc. of International Conference on Hardware/Software Codesign and System Synthesis*, 2008, pp.143-148

[12] M. Hassan et al., “Enabling RTOS simulation modeling in a system level design language,” in *Proc. of Asia and South Pacific Design Automation Conference*, 2005, pp.936-939

[13] A. Bouchhima, S. Yoo, and A. Jerraya, “Fast and accurate timed execution of high level embedded software using HW/SW interface simulation model,” in *Proc. of Asia and South Pacific Design Automation Conference*, 2004, pp.469-474

[14] A. Gerstlauer, H. Yu and D. Gajski, “RTOS modeling for system-level design,” in *Proc. of Design, Automation and Test in Europe,* 2003, pp.130-135

[15] S. Yoo, I. Bacivarov, A. Bouchhima, Y. Paviot, and A. Jerraya, “Building fast and accurate SW simulation models based on hardware abstraction layer and simulation environment abstraction layer,” in *Proc. of Design, Automation and Test in Europe*, 2003, pp.1530-135

[16] QEMU, Available: http://www.qemu.org

[17] VMware, Available: http://www.vmware.com

[18] Anton Chernoff et al., "FX!32: A Profile-Directed Binary Translator," *IEEE Micro*, vol. 18, no. 2, 1998, pp. 56-64