A Computational Reflection Mechanism to Support Platform Debugging in SystemC

Bruno Albertini, Sandro Rigo, Guido Araujo
Institute of Computing, Unicamp
1251 Albert Einstein – Campinas, Brazil
{sandro, bruno.albertini, guido}@ic.unicamp.br

Cristiano Araujo, Edna Barros, Williams Azevedo
Informatics Center, UFPE
Av. Professor Luiz Freire s/n – Recife, Brazil
{cca2, wtoa, ensb}@cin.ufpe.br

ABSTRACT

System-level and Platform-based design, along with Transaction Level modeling (TLM) techniques and languages like SystemC, appeared as a response to the ever increasing complexity of electronics systems design, where complex SoCs composed of several modules integrated on the same chip have become very common. In this scenario, the exploration and verification of several architecture models early in the design flow has played an important role. This paper proposes a mechanism that relies on computational reflection to enable designers to interact, on the fly, with platform simulation models written in SystemC TLM. This allows them to monitor and change signals or even IP internal register values, thus injecting specific stimuli that guide the simulation flow through corner cases during platform debugging, which are usually hard to handle by standard techniques, thus improving functional coverage. The key advantages of our approach are that we do not require code instrumentation from the IP designer, do not need a specialized SystemC library, and not even need the IP source code to be able to inspect its contents. The reflection mechanism was implemented using a C++ reflection library and integrated into a platform modeling framework. We evaluate our technique through some platform case studies.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General;
J.6 [**Computer Applications**]: Computer-aided Engineering—*Computer-aided Design (CAD)*

General Terms

Design, Verification

Keywords

Platform-based Design, Debugging, Computational Reflection, System Architecture

CODES+ISSS'07, September 30–October 3, 2007, Salzburg, Austria.

1. INTRODUCTION

The ever increasing system complexity has been motivating the creation of new design methodologies for several years now, culminating with the so called Electronic System Level (ESL) design and platform-based design. In this scenario, the exploration of several SoC architectural models is key to achieve application tuning and improved performance. This demands a platform simulation infrastructure capable of doing fast simulation both software and hardware, at a high level of abstraction. SystemC [3] has emerged as one of the most adopted system level design languages and Transaction Level Modeling (TLM) is being widely adopted as the most suitable technique for ESL design. One of the most appealing aspects of TLM is the possibility of reusing the whole platform infrastructure for both hardware and software development and verification [9].

The integration of verification into the design flow is very important in a TLM-based design methodology. For example, by guiding the simulation flow through certain corner situations by means of specific stimuli injection. This type of feature is very useful in order to increase verification coverage. In this paper, we propose a mechanism that relies on computational reflection to enable designers to, on the fly, interact with the platform simulation. This allows them to monitor and change signals or even IP internal register values, so as to explore, in a high level of details, what is really happening within each platform module. Our main goal is to improve the designers' ability to test corner cases during platform debugging, thus improving functional coverage of their verification techniques, without requiring any modification on the IPs SystemC code. Actually, the designer does not even require the source code of an IP to be able to apply the reflection mechanism to it, as we are going to explain further in the text. Our reflection mechanism was implemented using a C++ reflection library and integrated into a platform modeling framework called PDesigner [12], which enables the construction of SystemC platform models graphically, through an Eclipse plugin, thus facilitating platform simulation and exploration.

The remaining of this paper is organized as follows: Section 2 discusses some related work, Section 4 presents our platform simulation infrastructure, Section 3 introduces the reflection mechanism, Section 6 illustrates the application of the reflection technique through platform debugging case studies, and finally, Section 7 summarizes our conclusions.

2. RELATED WORK

The majority of the commercial ESL tools that allow functional description and simulation, enable support to some kind of debugging. This is usually achieved by library instrumentation, which implies a modified version of SystemC and TLM libraries tied to their development software.

Coware's Model Designer [5] is an Eclipse plugin that supports IP modeling using SystemC. By using Coware's SystemC library, it is possible to monitor process execution, event generation chain and all communication, including TLM. Some of the data can be modified on the fly, but the main purpose seems to be logging and test management. The Incisive family, from Cadence [4], has many introspection tools. The SystemC from this vendor supports dynamic assertions and code instrumentation. Incisive Simulator is a tool to automatically instrument code, aiming the automatic test generation using Incisive Scenario Builder. Mentor Graphics [11] seems to be investing in XML IP descriptions. Every IP has a description of its structure, including interface. This helps to generate tools for monitoring, since at least the exposed structure is always known. Nevertheless, it cannot be used to inspect internal registers.

Up to now, the work on the SystemC Verification Library (SCV) has focused on RTL verification features. We are aware that SCV developers are considering high-level verification features for future releases. There is no data introspection mechanism on the current version of SCV.

Lapalme et al [10] developed a platform design framework based on Easy.NET, an extension of .NET for hardware descriptions. They first identified the need of code introspection for debugging and verification purposes but, for them, it came at no cost since C# (from which .NET was derived) natively supports reflection. Déharbe and Medeiros [6] showed another way of doing the same thing using aspect oriented programming. The focus was instrumenting all the code to generate code metrics, but the same technique can be easily used for verification and debugging purposes. The instrumentation was made using the AspectC++ library, a known C++ extension for this programming paradigm. The main disadvantage is that IP designers must be aware that the IP will be accessed using this technique and write their code using the aspect oriented paradigm.

In this paper, we describe an introspection mechanism, based on computational reflection, for aiding designers during platform debugging in SystemC. The main advantages of our approach, if compared to the alternatives discussed in this section, are that we do not require code instrumentation from the IP designer, do not need a specialized SystemC library, and do not even need the IP source code to be able to inspect its contents. We clarify these concepts in the following sections.

3. COMPUTATIONAL REFLECTION

Computational reflection is the ability of representing the system within the system itself [8]. Some authors classify different reflection techniques considering the mechanism used to generate the additional data structure needed for achieving the introspection. Static reflection examines the source code of the reflected data structure at compile time and solve all necessary conflicts (i.e. type) before compilation. Dynamic reflection generates separated structures that can be compiled and linked with the code that will use the reflected data. This approach allows the code that uses the reflected data to be written in a generic way, without knowing the reflected data structure at compile time.

Static reflection is widely used by compilers to do memory optimizations, but it does not fit our needs. We need a reusable module that can be used to inspect and change any SystemC module. Dynamic reflection can be used to our purposes with the small overhead of generating the external data structures for every reflected module. Libraries that implement this kind of reflection do it by means of a dictionary. Although many recent languages like C#, .NET, and Java support some kind of reflection natively, C++ does not. Modern compilers generate internal reflection for memory optimization, but there is no way the end user can easily access those structures. There are many libraries that implement reflection, most of them based on user annotation or static reflection, but usually they do not provide all the necessary features to do dynamic reflection.

One of the requirements we wanted to include in our methodology is that the IP designer should not need to be aware that the IP will be inspected, meaning that designers will develop their SystemC IPs in exactly the same way they already do. Moreover, it should also be possible to inspect, through the reflection mechanism, IPs whose source code are not even available. These requisites make static reflection or user annotated strategies, where the reflection mechanism or the source code must be rewritten for every single IP, unsuitable for our purposes.

Aspect oriented programming is an interesting programming paradigm which can well suit data introspection. But it requires changing the programming paradigm in which designers develop their IPs. Each SystemC module where data introspection would be applied should be developed following the aspect oriented programming paradigm. As we see, this requisite has two major drawbacks. First, hardware designers may not be willing to learn and to change between different programming paradigms, seriously restricting the adoption of this technique. Second, all existent IPs need rewritten in order to make the data introspection technique applicable.

4. MODELING AND SIMULATION INFRASTRUCTURE

Debugging of SystemC models demands an environment that provides the user with good interactivity with the system. In this work, we adopted the PDesigner [12] framework as our main infrastructure. This is a modeling, simulation, and analysis framework for SystemC TLM based multiprocessor platforms. The framework abstracts the SystemC 2.2 library and simulator from the user that works as a platform integrator. The SystemC TLM component library available in PDesigner includes processor models designed with an architecture description language (ArchC 2.0), bus functional, cycle approximate, and cycle accurate models of AMBA and Avalon buses, and hardware IP cores. A screenshot of PDesigner is shown in figure 1, marked with letters `A, B` and `C` in each different view available to the user. The `A` view is the project management area with the project directories. In the `B` view is the component palette that allows the user to interact with the component library, the user can drag and drop components in the platform view (`C`). There it can be seen a platform with a MIPS processor, one jpeg decoder

IP, one SimpleBus bus, and two memories and their connections. On the bottom, it can bee seen a console window (D) that is part of the computational reflection tool to be further explained in the following sections.

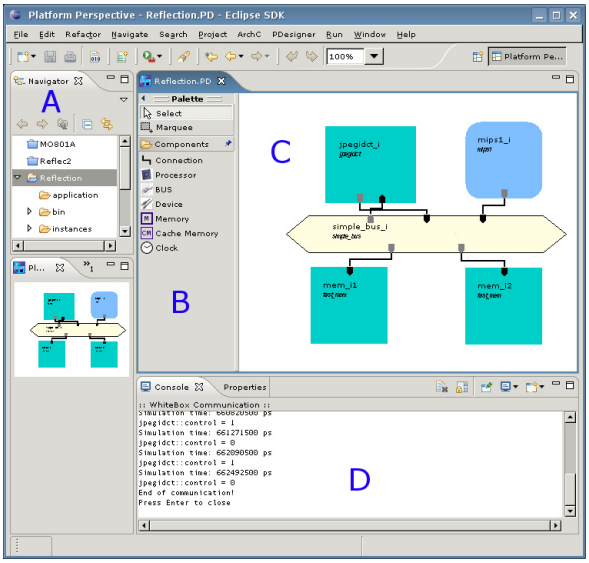

Figure 1: PDesigner Screenshot

An important feature of the framework is that it provides support for the SPIRIT 1.2 LGI interface [14] to integrate new tools. The LGI interface makes use of the SPIRIT schema files to allow the integration between the modeling and simulation tools with the analysis tool that interacts with the platform or with a specific component.

The PDesigner [12] architecture is depicted in figure 2. It can be divided in three distinct parts. The first part is a set of standard tools, languages, and simulation environment. The simulation environment is build upon the SystemC 2.2 library. SystemC allows SoC components to be modeled at different abstraction levels. For now, it supports functional, timed approximate, and cycle accurate component models. The connection and communication between the SoC components is based on the SystemC TLM 1.0 standard [9], providing simpler connections, easier protocol modeling and faster simulation. On top of the simulation and communication is the ArchC architecture description language [2] and tools. It is a SystemC like language used for processor architecture description and automatic generation of software tools like simulators, assemblers, and linkers. By using the ArchC ADL, it is possible to generate SystemC processor models instrumented with TLM interfaces.

The second part is a set of Eclipse [7] plugins that compose the core tools for modeling, simulation, and library management. The PArchC plugin enables processor architecture designers to model processors using ArchC and to export these processors to the component library. The distribution of SystemC TLM components other than ArchC processors is done using the IPZip plugin [1]. This plugin provides support for a component designer to distribute it without having to deal with the several SPIRIT schema files. This plugin implements a set of wizards that guides the component distribution. The distribution flow is depicted in figure 3. The result of the distribution is a zip file containing the

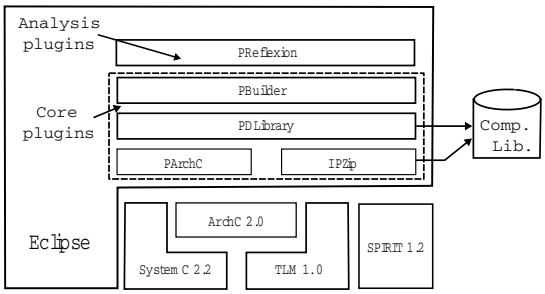

Figure 2: Platform Design Framework Architecture

component source code and the SPIRIT schema files, thus allowing the component to be imported into the framework component library. As a complement to the PArchC and IPZip plugins, PDLibrary is responsible for managing the component library by enabling users to easily add and remove SPIRIT distributed SystemC TLM components from it.

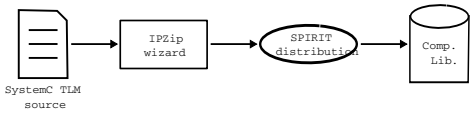

Figure 3: SystemC TLM Distribution Flow

The last element of the core plugins is PBuilder. It gives the user an environment for modeling and simulating SoCs. Its main purpose is to abstract the SoC designer of implementation details of the platform components. On one hand, it abstracts SystemC language details as the user drags-and-drop components in a graphical environment. This environment also provides a view for parameter tuning of the components. On the other hand, PBuilder transparently enables the connection between components that talk different protocols. The user connects master and slave protocol ports and the tool inserts the necessary wrappers, that perform the protocol conversion, from the library.

The third part is composed of analysis plugins. These plugins make use of the platform architecture information provided by PBuilder in order to offer analysis services to the user. Section 3 describes how we use the computational reflection concept and Section 5 explains the implementation of the mechanism developed in this work, which was integrated to the PDesigner framework.

5. PLATFORM REFLECTION IMPLEMENTATION

We evaluated several reflection libraries in order to implement our platform debugging mechanism. We ended up choosing the Reflex-SEAL [13] library, designed by CERN as a part of the ROOT project. The advantages over other libraries like CPPReflect, AReflection, or OpenC++ are mainly: non intrusive, semi automatic information gathering, no external dependencies except for the library itself, and there is no need to modify or replace the compiler.

The information gathering is done in two phases. First, GCCXML is used to parse the source code header file and to generate an object equivalent structure in XML. Second, the

XML file is parsed, generating a compilable dictionary that contains information about the object structure, like offset and type of the attributes and methods. All information gathering is done before the compilation and should be done once for every module to be reflected. Notice that the no source code file (.cpp) is touched and no annotation on any source code is required.

Programs that use the reflected data can access the generated dictionary using the Reflex-SEAL library. Since the dictionary has all the internal structure representation, just getting a pointer to the object is sufficient for any reflection operation. In this paper we are interested in attribute inspection for reading and writing, but calling methods and non virtual functions are also possible. The library has some aiding functions, like iterators over attribute lists and type information, very useful when writing generic code.

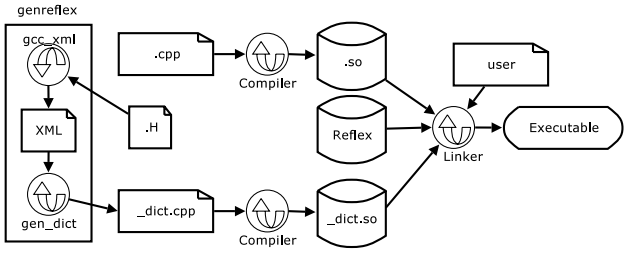

Figure 4: Reflex-SEAL Reflection Flow.

Figure 4 shows the reflection generation flow. Dictionary generation is performed by a script called `gen_reflex`, shipped with the library. This script calls GCCXML and generates the dictionary. `User` is the program that will use the reflected data, and linking it with the library and the compiled dictionary is sufficient.

The SPIRIT standard has a lot of interesting characteristics ruled by its specification, from IP interface to abstraction layers, but we are specially interested in one mechanism introduced by the SPIRIT team: the *WhiteBox*.

A WhiteBox is a module that can be plugged into any hardware description to inspect the state of the hardware. SystemC hardware descriptions are C++ classes. Data inspection can be done inspecting all inputs and outputs, represented by variables, as well as the register information, represented by object attributes and variables. Reflection can give us exactly this kind of information. The SPIRIT consortium does not specify the way that WhiteBox modules should be implemented, but its functionality. Our implementation consists in a SystemC module that has zero delay and communicates with the world by means of a socket and a simple protocol. It acts like any SystemC module, but it cannot be accessed by another module because it has no interface. When SystemC's kernel schedule the process of this module it uses the dictionary for walking through an instance of the reflected object (another SystemC module) and gather information.

The WhiteBox is generic and can be instantiated for any kind of SystemC module by passing a pointer to the object when calling the WhiteBox's constructor. A WhiteBox can handle just one IP instance, but the platform can have as many WhiteBox instances as required. At the first execution, it will generate an internal list of the attributes and allow the user to set conditions, like what are the attributes of interest and what to do with them. Actually, Whiteboxes support breakpoints, used to stop the simulation when one of the conditions is satisfied or just log any changes suffered by the attribute of interest. When a breakpoint condition is satisfied, the user is capable of observing the value, of changing the value, or of just continuing the simulation. Remember that the WhiteBox has zero delay, so stopping the simulation does not advance SystemC's simulation time. WhiteBox scheduling must be performed in such a way that it inspects the reflected module every simulation cycle where the module could change its state. It is accomplished by defining the WhiteBox sensitivity list carefully, making it sensible to the same events that can trigger the reflected module.

Using the WhiteBox is a pretty simple task. First, the user should reflect the desired module. This can be done automatically with the dictionary generator `gen_reflex` and the header file of the reflected IP, as previously mentioned. If the designer is using PDesigner as his capture tool, a right mouse button click over the desired IP should open a drop box containing the reflection option. Everything else is done internally by PDesigner.

Including the mechanism directly in SystemC descriptions is straightforward. Listing 1 shows a code snippet containing an example where a WhiteBox module is used to inspect an IP that is connected to a bus. Line 5 creates a clock signal, while lines 6-8 do the module instantiations. The clock is binded to the bus clock signal port (11). So, this is the same signal that must be used to trigger the WhiteBox module (line 10). Notice that the WhiteBox also receives a string containing the reflected object class name (line 7), used for dictionary query, and a pointer to the IP instance.

Listing 1: WhiteBox Usage

```
1 #include "`ip.H"
2 #include "`whitebox.H"
3 int sc_main(int ac, char *av[]) {
4   // Common clock
5   sc_clock bus_clock;
6   ip myip("ip",3,0x300500,0x3FFFFF);
7   whitebox<ip>
8         wb("wb","ip",&myip,6000);
9   // Binding with same sensitivity list
10  wb.clock(bus_clock);
11  myip.clock(bus_clock);
12 };
```

The final step is to connect the WhiteBox with a socket interface compliant with its protocol. It is a simple handshake protocol that we omit here for the sake of saving space. By default, the WhiteBox will listen for connections at port 6000, but this can be configured at WhiteBox instantiation, which is useful when reflecting more than one module at the same time. PDesigner has the ability to connect to the WhiteBox module and to show reflected information. Figure 1 shows a WhiteBox output, in the console window (`D`), as it is provided to the user in the PDesigner interface.

6. CASE STUDIES

Multi-IP Platform

The platform used for the first case study is composed by a PowerPC (PPC) functional model, generated by ArchC,

a programmable MMU (Memory Management Unit) for address translations, an AMBA bus with an arbiter, and a functional memory. The processor controls an IP chain, using it to do numeric computation, and two timers that feed the processor through a programmable interrupt controller. For this example, the PPC uses the IPs as random number generators in order to fill up vectors for its application. Figure 5 shows an overview of the platform.

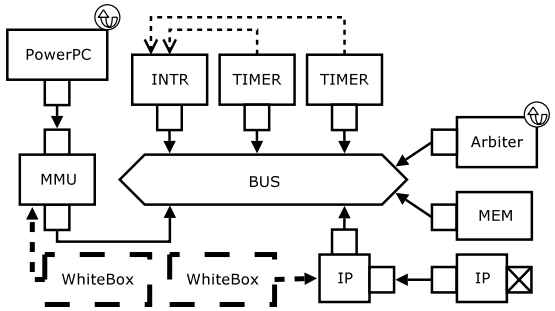

Figure 5: Diagram of the Case Study Platform.

The main point about this platform is its memory address space, which totalizes 65535 bytes. The physical memory has 9MB of real address and IPs (including timers and interrupt controller) are mapped to addresses between 9MB and 10MB. The MMU is responsible for paging and redirecting issues. Offset redirection is done automatically, but paging should be programmed by the processor. Accesses to the memory region from 0 to 2MB and from 3 to 10MB are redirected to real memory through MMU paging. Memory addresses from 2MB to 3MB are redirected, without paging, to IP addresses from 9MB to 10MB. One important point is that designers of this platform did not have access to any source code except for processor. For the remaining platform modules, all they had were the prototype headers and object files (.H and .o files).

The first execution revealed that the memory application vector had been filled up with zeroes. It could appear that the IPs were not working properly at first look, but when IPs were tested in isolation from the platform they were working fine. So the problem only appeared when they were connected together into the platform. The usual verification flow for SystemC models takes the designer to a debug tool like GDB but, since we do not have the source code, there is no guarantees that the object file will work with GDB. Moreover, it is known that SystemC debugging can get really annoying if GDB falls into SystemC's library code, instead of just following the application code.

Our approach consists in using reflection to dynamically inspect the register values of all SystemC modules that take part of the process. This methodology showed that the IPs were not receiving any requests at all and so, we reflected the MMU module.

The inspection of MMU registers at runtime showed that the MMU was paging the IP mapped addresses (from 2 to 3MB). Since this space is physically tied to the IPs, the accesses to this area should not be paged, but just added to an offset to redirect it to a real IP address (from 9 to 10MB). The problem was reported to the MMU developer, who sent a patched MMU version that worked in a first run. All the inspection process took less than one hour.

DJPEG architecture

The second case study was taken from a real world example. The platform was a PowerPC running the Mediabench JPEG decoder. As a first design cut, the designer ran a full software platform composed by a PowerPC, bus, and memory, for profiling purposes. He noticed that the integer discrete cosine transformation was taking almost a half of the processing time, so it made sense to design an IP to do that job.

The designer created a functional description of a specialized IDCT IP and connected it to the platform using a TLM interface. Inside the IP, a SystemC process receives the incoming TLM requests and monitors a control register, awaiting for a non-zero value to trigger the IDCT computation. In fact, the IP acts like a slave for the processor, which programs the IP registers and sets the control register to signal the IP. Once the IP is triggered, it changes to master behavior, racing with the processor for accessing the memory (and consequently for the bus). After all computation is done, the IP sets its control register back to zero, signaling to the processor that the job is done, and returns to slave state. This platform is shown in figure 6.

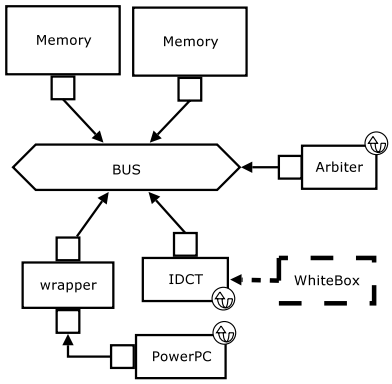

Figure 6: Diagram of the JPEG Platform.

The reflection mechanism was used in this platform with two purposes. First, the designer used it to do timing analysis. As the control register holds true when the IP is working, and zero when it is in slave state (sleeping or no job state), the designer sets an "on change" breakpoint on this register. The WhiteBox outputs all transitions from zero to one and from one to zero. As all the outputs receive SystemC's internal simulation time-stamp, the timing between two consecutive executions can be easily determined. This information was used to conclude that the IP was too fast for bus interleaved usage. The processor should stay in a busy/wait mode or the IP must be duplicated to allow a parallel execution or double buffering, for example.

Second use on this platform was to debug parameter passing. The result image was weird when the IP was first used, so the designer reflected the IDCT IP to inspect its registers. One of the parameter registers is a vector address in memory. The memory was originally allocated by the application, but since we do not have any dynamic allocation on this small system, a hard coded vector was used. The inspection of this register and the incoming TLM transaction packet showed that the address arriving to the IP was not the same as the one sent by the processor. The PowerPC is a big endian

processor, but the ELF files were not converted when loading, so the first position of the vector was right, but after a few iterations the sum done by processor was somewhat far away from the vector address space. In this platform, we do not have any operating system to complain about illegal memory accesses. It was easy to devise this problem using a WhiteBox to reflect the data from both the register and the TLM packet variable, showing timestamp related transactions side by side. After applying a conversion function and correcting some typecasting, all access were corrected and the final decoded image was the expected.

The DJPEG platform was also used to a performance evaluation. Our goal was to estimate the performance downgrade imposed to the platform simulation due to the WhiteBox insertion. Table 6 shows the performance statistics collect for three different configurations of this platform. The first configuration includes the IDCT IP sharing the job with the PPC processor. The same scenario as in Figure 6, excluding the WhiteBox. The second configuration includes a stub IP with an empty SystemC process, added with the sole purpose of including one more SystemC process to be scheduled during simulation. The third platform is the one depicted in 6, including the WhiteBox reflecting the IDCT IP.

Configuration	KIPS
IP	774.37
IP + Stub	725.87
IP + WhiteBox	726.42

Table 1: DJPEG Platform Performance.

We can see from the first two configurations that adding the stub IP caused a simulation speed downgrade due to the overhead of having one more SystemC process to schedule. The last two lines of the Table show us that substituting the stub IP for the WhiteBox module caused a similar performance downgrade. The simulation performance overhead imposed by the WhiteBox insertion is similar to the overhead on having one more empty IP or empty SystemC process in the platform. Speeding up SystemC's internal process context switching and synchronization mechanism would decrease the downgrade imposed by the reflection mechanism significantly.

7. CONCLUSIONS

This paper introduces a new platform debugging mechanism based on computational reflection to achieve data introspection in SystemC hardware modules. Our mechanism enables designers to guide the platform simulation flow, by observing and changing signals and IPs internal register values, allowing the injection of specific stimuli. The result is an improved ability to cover corner cases in the platform simulation, ending up increasing the functional coverage of the whole verification process. The key advantages of our approach are that it does not demand any special action or IP code preparation from the designer, nor a specialized SystemC library, and not even the IP source code files in order to enable the data introspection debugging technique. The reflection mechanism requires only the source header file (.h) and the object file (.o) for each platform module to be inspected.

The mechanism is built upon the open source ReflexSEAL reflection library and was integrated to the PDesigner platform modeling and simulation framework. However, it is important to notice that both the introspection technique and the computational reflection mechanism are independent from the framework. It could be integrated to any SystemC-based platform development environment. Moreover, the impact of using the reflection technique on the simulation performance is small, equivalent to having an extra empty SystemC process in the system.

8. ACKNOWLEDGEMENTS

The authors would like to thank CAPES (processes number 0018058 and 0326054) and CNPq (processes number 55.2117/20021, 132916/20053, and 477457/20061) for funding this project.

9. REFERENCES

[1] C. Araujo, E. Barros, M. Almeida, and G. Araujo. IPZip - An IP Distribution Framework. *Proceedings of the IP-SOC 2006*, pages 259–264, Dec. 2006.

[2] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros. The ArchC Architecture Description Language. *International Journal of Parallel Programming*, 33(5):453–484, Oct. 2005.

[3] D. C. Black, J. Donovan, B. Bunton, and A. Keist. *SystemC: From the Ground Up*. Kluwer, 2004.

[4] Cadence. Website, Jan. 2007. `http://www.cadence.com`.

[5] Coware. Website, Jan. 2007. `http://www.coware.com`.

[6] D. Déharbe and S. Medeiros. Aspect-oriented design in systemC: implementation and applications. In *SBCCI '06: Proceedings of the 19th annual symposium on Integrated circuits and systems design*, pages 119–124, New York, NY, USA, 2006. ACM Press.

[7] Eclipse - an open development platform. `http://www.eclipse.org`, 2007.

[8] J. Ferber. Computational reflection in class based object-oriented languages. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 317–326, New York, NY, USA, 1989. ACM Press.

[9] Frank Ghenassia, editor. *Transaction-Level Modeling with SystemC*. Springer, 2005.

[10] J. Lapalme, E. M. Aboulhamid, and G. Nicolescu. A new efficient EDA tool design methodology. *Trans. on Embedded Computing Sys.*, 5(2):408–430, 2006.

[11] Mentor graphics. Website, Jan. 2007. `http://www.mentor.com`.

[12] PDesigner. `http://www.pdesigner.org`, 2007.

[13] S. Roiser and P. Mato. The SEAL C++ reflection system. In *CHEP '04: Presented in the Computing in High Energy and Nuclear Physics congress (CHEP'04)*. CERN, Sept. 2004.

[14] SPIRIT Consortium. `http://www.spiritconsortium.org`, 2007.