

Understanding and Extending SystemC User Thread Package to IA-64 Platform.

J. Vennin
Prosilog
8, rue traversiere
95000 Cergy, France
vennin@prosilog.com

S. Meftali
LIFL - USTL
Cite Scientifique, 59655
Villeneuve d'Ascq cedex, France
meftali@lifl.fr

J.L. Dekeyser
LIFL - USTL
Cit  Scientifique, 59655
Villeneuve d'Ascq cedex, France
dekeyser@lifl.fr

ABSTRACT

In this paper, we present an extension of the SystemC simulator in order to allow its execution on an IA-64 platform. Our approach relies on adding to SystemC a new user thread package in a simple way. The proposed user thread mechanism is based on the *ucontext* primitives, and can be integrated or updated in SystemC easily.

The effectiveness of this approach is shown on a concrete size example composed of two masters and two slaves connected to an AMBA bus.

Keywords

SystemC, context switching, 64 bits architecture

1. INTRODUCTION

SystemC[4, 1] appeared in 1999 as a new language for hardware description in system design. It is a C++ library which allows hardware and system-level modeling. Since its creation, SystemC became a more mature simulator with new functionalities. In fact, the first version was limited to RTL¹ system design only. With the 2.0 version, abstracts layers have been added to SystemC in order to get more and more design abstraction[7]. These enhancements brought transaction level modeling capability to the initial language specification.

The few last years also have been marked by the emergence of new embedded applications in several domains such as image processing, games, etc. The particularity of these applications is that they handle big amount of bulky data, and this makes usually their representation on 32 bits architectures very tedious. IA-64 (Intel Architecture-64) is a 64 bits architecture developed by Intel and Hewlett Packard for processors such as Itanium. It is an architecture where data are stored in chunks of sixty-four bits each.

¹Register Transfer Level

The use of these modern 64-bits architectures as a simulation support for this kind of applications can have a great value in terms of performances[5]. Unfortunately, no tool to our knowledge allows the execution of the SystemC simulator on an IA-64 platform. So the objective of this work is to propose a new user thread package within SystemC that allows the execution on the IA-64 architecture in a simple way.

The development of this new user thread package specifically for IA-64 architecture is motivated by two factors: accuracy and simulation speed. Thus, in many embedded applications as in complex mathematical computations, we usually need a high accuracy both in data representation and computation results. This class of applications can be of course modeled and executed on 32 bits classical architectures. In this case, the 64 bits representations are just emulated and this can increase consequently the simulations time.

This paper is structured in six paragraphs. We will first present in section 2 the SystemC kernel and the kernel and user thread concept. Next, we will describe in details the abstract layer used by SystemC and its different implementations in section 3. After that, in section 4, we will show how a new thread package is created in order to obtain a fast simulation engine on the IA-64 architecture. This leads to section 5 showing the performances of our new user thread package using a significant example. And finally, we conclude this paper in section 6.

2. KERNEL AND USER THREAD

In [3], the basic unit of the SystemC functionality is called a *process*. This name *process* is commonly used to designate concurrent behavior mechanism. The reason of this basic unit choice in SystemC is that in electronic systems, all operations are done in parallel and are often concurrent. SystemC discerns three types of process, the *SC_METHOD* process, the *SC_THREAD* process and the *SC_CTHREAD* process. The user thread level mechanism is only used by *SC_THREAD* and *SC_CTHREAD* processes. Only this type of SystemC process are taken into account in this paper. We make no distinction between the *SC_THREAD* and *SC_CTHREAD*, since the only difference between them is that an *SC_CTHREAD* is sensitive on a clock whereas an *SC_THREAD* process can be sensitive on any event.

To implement parallel and concurrent mechanisms we need

to use specific primitives of our operating system. These must have context switching capabilities, which can be implemented in two ways, using *kernel level threads* or using *user level threads*[6].

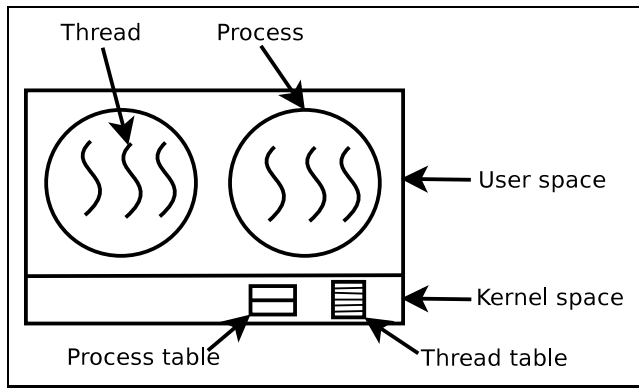


Figure 1: Kernel thread

If we use kernel level threads, the operating system will have a descriptor for each thread belonging to a process and it will schedule all the threads. This method is commonly called *one to one*. Each user thread corresponds to a kernel thread (figure 1). There are two major advantages around this kind of thread. The first one concerns switching aspects; when a thread finishes its instruction or is blocked, another thread can be executed. The second one is the ability of the kernel to dispatch threads of one process on several processors. These characteristics are quite interesting for multi-processor architectures. However, thread switching is done by the kernel, which decreases performances.

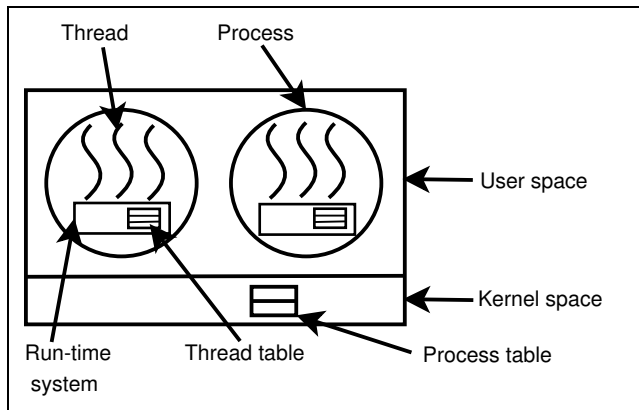


Figure 2: User thread

User level threads are implemented inside a specialized library that provides primitives to handle them. All information about threads is stored and managed inside the process address space(figure 2). This is called *many to one*, because one kernel thread is associated to several user threads. Its has some advantages: The first is that is independent of the system, thus, it runs faster than context switching at kernel level. The second comes from the scheduler that can be chosen by the user in order to manage a better thread execution. Nevertheless, if a thread of a process is jammed, all other threads of the same process are jammed too. Another

disadvantage is the impossibility to execute two threads of the same process on two processors. So, user level thread is not interesting in multi-processor architectures.

For SystemC, the user level thread mechanism is used in order to reach maximum performances. In fact, a SystemC module can have several threads (SC_(C)THREAD) and within the same design project, we could use hundreds of SystemC modules. Thus, it's not conceivable to use kernel thread level in a simulator to simulate parallel and concurrent mechanism.

3. SYSTEMC USER THREAD PACKAGES

As we explained in the previous sections, SystemC uses user level thread mechanism. In this section we describe the abstract layer used by SystemC and the different implementations of this abstract layer

3.1 The user level thread abstract layer

SystemC developers have made a great job around the user level thread. They have defined abstract classes with the required primitives in order to permit context switching. The two most important ones are: *sc_cor* and *sc_cor_pkg*. The first one is an abstract class allowing the creation of a new user thread package. The second one, represents an abstract class for one user thread. The following subsections present some aspects of the *sc_cor_pkg* and *sc_cor* abstract classes. The *wait* statement is also explained.

3.1.1 Basics of sc_cor_pkg abstract class

The SystemC coroutine package allows the creation of a *sc_cor* object, the abortion of the current co-routine and the yielding to the next co-routine. The abstract class is defined as follows:

```
class sc_cor_pkg {
public:
    ...
    virtual sc_cor* create( size_t stack_size,
                          sc_cor_fn* fn, void* arg );
    virtual void yield( sc_cor* next_cor );
    virtual void abort( sc_cor* next_cor );
    ...
};
```

The *create* method allocates a *sc_cor* object with some parameters: the desired stack size for the new user thread, the callback function, which will be called inside the *yield* or the *abort* method and an argument for the callback function.

The *yield* method allows to process the function associated to the *next_cor* parameter.

The *abort* method stops the execution of the current process and continues the execution of the function associated to the *next_cor* parameter.

3.1.2 Basics of sc_cor abstract class

The *sc_cor* abstract class has no particular functionalities. Each user thread instance is a class, which inherits the *sc_cor* abstract class.

3.1.3 The wait statement

Any thread API², usually provides some functions to suspend a thread depending on one or several conditions. For this purpose, SystemC defines the *wait* statement, which is implemented in different ways: a thread can wait for a specific events, or wait during a specific time. When the *wait* function is called, the *yield* method of the *sc_cor_pkg* is executed in order to switch to the next thread.

3.2 Existing implementation of SystemC thread packages

SystemC 2.0.1 version has only two thread packages: the QuickThread packages and the fiber thread packages. The incoming SystemC 2.1 has a new user thread package which is based on the *pthread* POSIX³ library. In the following section, we explain briefly each user thread package.

3.2.1 QuickThread package

QuickThread is a toolkit for building threads packages. It works on several architectures:

- 80386 family,
- 88000 family,
- DEC AXP (Alpha) family,
- HP-PA family,
- KSR,
- MIPS family,
- SPARC V8 family,
- VAX family.

SystemC uses the QuickThread toolkit as a user thread package; it only runs on Unix like operating systems. The QuickThread toolkit does not work on IA-64 architecture, but it is still possible to extend it to support additional platforms. However, this requires knowledge of the target architecture and supposes the writing of dedicated assembler code routines to handle context switching and relies on it.

3.2.2 Fiber Thread package

This package is only targeted for the Windows operating system. Its kernel API offers primitives for context switching. Microsoft defines a *Fiber* as a unit of execution that must be manually scheduled by the application. SystemC makes benefit of this features in order to run it under Windows operating system.

3.2.3 Pthread POSIX package

This package uses the pthread POSIX library which is available on most operating system. It allows compiling and to using SystemC on almost all operating system. The applications portability is very important, however there is a drawback: using *pthread* means simulating a concurrent

²Application Programming Interface.

³Portable Operating System Interface for uniX.

mechanism with *SC_THREAD* or *SC_CTHREAD*, which in turn implies creating a kernel thread. Thus, for instance, if we have a design project with hundreds concurrent mechanisms we will have also hundreds kernel threads created. This thread package can not be used to achieve better simulation performances[].

4. HOW TO CREATE OUR USER THREAD PACKAGE

We have seen in section 3.2.1 that the QuickThread package doesn't work on IA-64 architecture. We can compile SystemC with the pthread package in order to compile under IA-64 but we want fast simulation. Thus, it is necessary to create a new thread package in order to have a fast simulation engine under the IA-64 architecture.

The following sections show our choice around the *ucontext* primitives and how to create a user thread package.

4.1 ucontext primitives

The *ucontext* based functions are part of XPG4 standard and hence are present only in XPG4 compliant libc's, such as GNU⁴ Libc (glibc). Using these as a basic building blocks, a full-fledged multi-threading library can be designed. The functions inside the *ucontext* are:

void makecontext(ucontext_t *ucp, void *func(), int argc, ...): Creates a new context structure and sets the function that must be executed and its parameters when you call a *setcontext* on the *ucp* variable.

int getcontext(ucontext_t *ucp): Initializes the structure pointed by *ucp* the current active context.

int setcontext(const ucontext_t *ucp): Restores the user context pointed by the *ucp*.

int swapcontext(ucontext_t *oucp, ucontext_t *ucp): Saves the current context in the structure pointed by *oucp*, and then activates the context pointed by *ucp*.

We used this set of functions to create our own context switching package for SystemC.

4.2 Our own context switching package

In order to create our own context switching package, we need to inherit from *sc_cor_pkg* and *sc_cor* classes. The following sections show a part of our implementation.

4.2.1 sc_cor_pkg_ucontext

We describe here the implementation of the *sc_cor_pkg_ucontext*, which inherits from *sc_cor_pkg*.

```
class sc_cor_pkg_jmp : public sc_cor_pkg {
public:
    sc_cor_pkg_ucontext( sc_simcontext* simc )
        : sc_cor_pkg( simc ) {}

    virtual ~sc_cor_pkg_ucontext() {}
};
```

⁴Gnu is Not Unix.

```

    sc_cor* create( size_t stack_size,
                  sc_cor_fn* fn, void* arg ) {
    sc_cor_ucontext * cor = new sc_cor_ucontext;
    ucontext_t * ctx = &cor->m_ctx;
    makecontext (ctx, (void (*) (void))fn, 1, arg);
    return cor;
}

void yield( sc_cor* next_cor ) {
    sc_cor_jump* new_cor =
    static_cast<sc_cor_jump*>( next_cor );
    sc_cor_jump* old_cor = curr_cor;
    curr_cor = new_cor;
    if( swapcontext (
    &old_cor->m_ctx, &new_cor->m_ctx) != 0)
    std::cout << errno << std::endl;
}

void abort( sc_cor* next_cor ) {
    sc_cor_jump* new_cor =
    static_cast<sc_cor_jump*>( next_cor );
    sc_cor_jump* old_cor = curr_cor;
    curr_cor = new_cor;
    if( swapcontext (
    &old_cor->m_ctx, &new_cor->m_ctx) != 0)
    std::cout << errno << std::endl;
}
};

```

We can see that the integration of *ucontext* inside SystemC is quite easy. The *create* method uses the *makecontext* function in order to create a context. The *yield* and *abort* methods have the same implementation and use the *swapcontext* function in order to stop the current context and to activate the new context.

4.2.2 *sc_cor_ucontext*

The *sc_cor_ucontext* class inherits from the *sc_cor* class. The context representation is realized by this class. In fact, this class is characterized by two important attributes. The first one, is the *m_ctx* that is *ucontext_t* type. This field allows storing information about the context switching. The second one, is the *m_buf* which stores all data of the context. Its size is determined by the *SC_DEFAULT_STACK_SIZE* macro.

```

class sc_cor_ucontext : public sc_cor {
public:
    sc_cor_ucontext () {
    getcontext (&m_ctx);
    m_ctx.uc_stack.ss_sp = m_buf;
    m_ctx.uc_stack.ss_size = SC_DEFAULT_STACK_SIZE;
    m_ctx.uc_link = NULL;
}

    virtual ~sc_cor_jump() {}

    ucontext_t m_ctx;
protected:

```

```

    char m_buf[SC_DEFAULT_STACK_SIZE];
};

```

In this class we have specialized the constructor. It gets the current context and stores it inside the *m_ctx* variable. Next, we initialize the buffer pointed by this variable.

5. EXPERIMENTAL RESULTS

In order to test the performances and the functionality of this new user thread package for SystemC, we decided to compare all the user thread packages of SystemC version 2.1beta10, on an Intel 32 bits architecture. The computer used is a Pentium 4 at 2GHz. The chosen system has four IP components: two masters and two slaves connected to a *AMBA AHB CLI*[2] controller(figure3). The arbiter ensures that only one master has an exclusive bus access at a time. The decoder selects a slave depending on the authorized master address. During the simulation, the master 1 sends some data to slave 2, the master 2 send some data to slave 1.

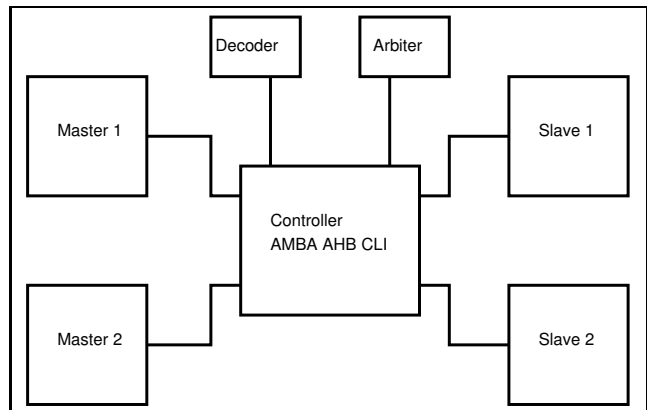


Figure 3: 2 Masters and 2 Slaves

QuickThread	PosixThread	UContext
5m28.190s	doesn't work	9m9.720s

Table 1: Simulation time on IA 32

Table 1 reports three test cases. The first observation is that the new POSIX thread package coming with the new SystemC version 2.1 beta 10 doesn't work for our system. The second, is that our new user thread package is not better than the QuickThread package. This can be explained by the fact that the *ucontext* primitives are implemented in C and assembler languages with some of the logic provided via system calls. They are also designed to work within signal framework. So, the design of *ucontext* primitives leads to some performance overhead.

The simulation has been executed on a 64 bits Intel architecture, an Itanium 2 with 4 processors at 1.2GHz. Table 2 shows the result of the launched simulation. Of course, the simulation was executed on one processor. The obtained simulation time is worth than the simulation time on the 32 bits Intel architecture. This difference is due to the clock frequencies of the two systems that were not the same. Another explanation is that SystemC was not optimized for a

64 bits architecture and this is why we obtain a low performance. However, even if the simulation time is longer, it is interesting to use a 64 bits architecture to simulate system which needs a lot of memory and accuracy.

QuickThread
12m33.440s

Table 2: Simulation time on IA-64

6. CONCLUSIONS AND FUTURE WORK

We have added our context switching package inside SystemC, to integrate it, some kernel modification was needed. To compile SystemC for the IA-64 architecture lots of job was required. The main problem to compile for this architecture was the *sc_fx_number* part, which required a huge amount of code lines.

We have shown in this paper how to extend the user thread mechanism integrated inside SystemC. The proposed user thread mechanism based on the *ucontext* primitive is not optimized yet, but it allows SystemC to run on a 64 bits Intel architecture. Another solution would be to extend the QuickThread library by adding to it the assembler code for context switching.

Our future work will focus on implementing a multi-threaded version of SystemC that will make benefit of multi-processors architectures and even run on dispatched SystemC clusters.

7. REFERENCES

- [1] Guido Amount. SystemC standard. In *Proc. of ASP-DAC 2000*, pages pp.573–577, Yokohama, Japan, January 2000.
- [2] A. Cochrane, C. Lennard, K. Topping, S. Klostermann, N. Weyrich, and K. Ahluwalia. AMBA AHB cycle level interface (AHB CLI) specification, 2003.
- [3] Thorsten Grotker. *System Design with SystemC*. Kluwer Academic Publishers, 2002.
- [4] The Open SystemC Initiative. <http://www.systemc.org>.
- [5] Intel. Highly optimized mathematical functions for the intel itanium architecture, June 2003.
- [6] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *Proceedings of the 13th ACM Symposium on Operating Systems Principle*, pages 110–121, Pacific Grove, CA, 1991.
- [7] Preeti Ranjan Panda. SystemC: a modeling platform supporting multiple design abstraction. In *ISSS*, pages 75–80, 2001.