Coroutine-Based Synthesis of Efficient Embedded Software From SystemC Models

Weichen Liu, Jiang Xu, *Member, IEEE*, Jogesh K. Muppala, *Senior Member, IEEE*, Wei Zhang, *Member, IEEE*, Xiaowen Wu, and Yaoyao Ye

***Abstract*—SystemC is a widely used electronic system-level (ESL) design language that can be used to model both hardware and software at different stages of system design. There has been a lot of research on behavior synthesis of hardware from SystemC, but relatively little work on synthesizing embedded software for SystemC designs. In this letter, we present an approach to automatic software synthesis from SystemC-based on coroutines instead of the traditional approaches based on real-time operating system (RTOS) threads. Performance evaluation results on some realistic applications show that our approach results in impressive reduction of runtime overheads compared to the thread-based approaches.**

***Index Terms*—Performance, software synthesis, SystemC.**

I. Introduction

SystemC [1] is a C++ class library for system and hardware design. It provides an event-driven simulation kernel for discrete event simulation. SystemC can be applied to system-level modeling, architectural exploration, performance modeling, software development, functional verification, and high-level synthesis [2]. Embedded software generated from SystemC usually runs on a resource-constrained platform with a lightweight real-time operating system (RTOS), or even runs without a RTOS. For code generation from SystemC, it is important to minimize the software running overheads on processor and memory resource usage. Software portability among various embedded systems is another important concern. In this letter, we develop an automatic code generation technique based on protothread, a platform-independent, and stackless coroutine to reduce the time and memory overhead of the generated code. The synthesized software is efficient in terms of performance and resource consumptions, and does not need any RTOS support. A flexible extension to the protothread prototype implementation is proposed to enhance this feature.

The rest of the letter is structured as follows: we present the coroutine-based protothread technique in Section II; the synthesis technique in Section III; discussions of related work in Section IV; performance evaluation results are given in Section V; and conclusions in Section VI.

Manuscript received December 04, 2010; accepted January 31, 2011. Date of publication February 10, 2011; date of current version March 23, 2011. This work was partially supported by Hong Kong SAR Grants GRF 621101 and RGC 617907. This manuscript was recommended for publication by S. K. Shukla.

W. Liu, J. Xu, J. K. Muppala, X. Wu, and Y. Ye are with the Hong Kong University of Science and Technology, Hong Kong, China (e-mail: weichen@ust.hk; eexu@ust.hk; muppala@ust.hk; wxxaf@ust.hk; yeyaoyao@ust.hk).

W. Zhang is with the School of Computer Engineering, Nanyang Technological University, Singapore (e-mail: zhangwei@ntu.edu.sg).

Digital Object Identifier 10.1109/LES.2011.2112634

TABLE I
Protothread Definition

```
#define PT_THREAD(name_args) char name_args
#define PT_BEGIN(pt) { char PT_YIELD_FLAG = 1;  \
            LC_RESUME((pt) -> lc)
#define PT_WAIT_UNTIL(pt, cond) do { LC_SET((pt) -> lc);  \
            if(!(cond)) { return PT_WAITING; }   } while(0)
#define PT_END(pt) LC_END((pt) -> lc); PT_YIELD_FLAG = 0;  \
            PT_INIT(pt); return PT_ENDED; }
```

II. Protothread

Protothread [3] is a programming abstraction that makes it possible to write event-driven programs in a thread-like style with a very low memory overhead. It significantly reduces the complexity of event-driven programs written with state machines. The execution time overhead of protothread is on the order of a few processor cycles.

Protothread makes it possible to exit a subroutine in the middle of its execution. For the next time of calling the same subroutine, it can continue running from the break point. With protothread, programmers can write code with multithreading in mind, but use protothread to replace real thread, and the generated software will run in a single thread from the system's point of view. This makes programming easy, largely reduces the context switching overhead generated by thread switching, and significantly improves the execution efficiency of the target software.

Protothread is a user-level thread library, which is lightweight with low runtime overhead, since context switches do not necessarily involve the OS kernel. The memory overhead is only two bytes per protothread, compared to several kBs for a regular POSIX thread [4]. It does not need any OS support and can be implemented on top of a single-process, single-threaded OS, or even bare hardware without any OS. The protothread library is based on a set of C macro definitions. The examples of prototype implementations of protothread is shown in Table I.

A. Applying Protothread to SystemC

In SystemC, there are two kinds of function declarations: SC_METHOD and SC_THREAD (SC_CTHREAD is treated as an equivalent SC_THREAD). SC_METHOD is a run-to-completion method call, so we can map it to a regular

TABLE II
SC_THREAD TO PT_THREAD TRANSFORMATION

void Func() {		PT_THREAD(Func(struct pt *pt)) {
		PT_BEGIN();
(. . .)		(. . .)
wait(event1);	=>	PT_WAIT_UNTIL(pt, event);
(. . .)		(. . .)
event2.notify();		event2.notify();
}		PT_END(); }

TABLE III
ENHANCE OF PROTOTHREAD FOR AUTOMATIC PROTOTYPE SELECTION

#ifdef LC_INCLUDE		#if defined(__GNUC__) ||defined(__GNUCPP__)
#include LC_INCLUDE		#include "lc-addrlabels.h"
#else	=>	#else
#include "lc-switch.h"		#include "lc-switch.h"
#endif /*LC_INCLUDE */		#endif /*LC_INCLUDE */

software subroutine in a straightforward manner. SC_THREAD is used for describing cyclic operations, so it is usually written using a while loop. We present a technique to map each SC_THREAD to a PT_THREAD, as shown in Table II. We integrate protothread into the SystemC kernel with these macro redefinitions. All SC_THREAD instances are implemented by protothread without impact to the rest of the code.

The transformed code only depends on the C preprocessor. Software generation does not require extra tools in the compilation tool chain. The resulting software is fully portable across all systems with/without RTOS support. However, the current implementations contain two important limitations: automatic local variables are not saved across a blocking wait statement; and C switch statements cannot be freely intermixed with protothread-based code. These problems are caused by the protothread mechanism. They can be solved by implementing protothread as a special precompiler for the software synthesis procedure or by integrating protothread into existing preprocessor-based languages and C language extensions such as nesC. We will discuss solutions to these issues in the following sections.

B. Improving Protothread for Automatic Prototype Selection

One limitation of our approach is that source code with C switch statement cannot be correctly transformed when the C switch statement is used to implement protothread. However, it can be avoided by using the GCC labels-as-values C extension [3]. We recommend SystemC programmers use GNU compiler [5] to compile their source code, so that this limitation can be overcome and C switch statement can be used freely during programming.

Protothread does not support automatic selection between prototype implementations with the C switch statement and the GCC labels-as-values C extension. We enhance protothread for automatic judgement and decision of which prototype to use according to the used precompiler, and make this procedure transparent from programmers. As shown in Table III, the common predefined macros "_GNUC_" or "_GNUCPP_" are checked to determine if the SystemC program is compiled in the GNU environment [5], and the prototype implementation of protothread with the GCC labels-as-values C extension is preferred to be chosen if possible, which uses line number of a program as the parameter of the goto statement for thread entry. Otherwise, the prototype with the C switch statement is used with the programming limitation. With this improvement, the protothread implementation is more complete, and it benefits SystemC software generation for improved flexibility.

III. SOFTWARE SYNTHESIS WITH PROTOTHREAD

Many researchers work on improving the execution speed of SystemC simulation [6], [7] since in general memory is sufficient in a simulation environment. However, things are different for embedded software generation, because the generated software usually runs on a lightweight RTOS, who has limited computation power and memory resource. Therefore, a performance and memory efficient approach is essential for embedded software generation. We present an approach using protothread, a platform-independent and stackless coroutine, for software synthesis to reduce multithreading overheads on time and memory.

Applying protothread has two major benefits. First, coroutines in SystemC simulation kernel is dependent to the design platform. Typically, the simulation kernel makes a decision on the type of thread according to the running OS. For example, Fiber is automatically used to implement a SC_THREAD on a Win32 platform, and it is hard for the users to switch to other types like POSIX or QuickThread [1]. This is bad for software generation, since we cannot guarantee the software is designed and executed both under the same OS, especially when the software is run on a RTOS that does not support the thread type used for generating it. Kernel-level threads, like POSIX, can bring more overheads since OS is involved in the context switching. The low efficiency is verified by the experimental results in Section V. Second, multithreading methods provided by SystemC are not memory friendly: either user-level threads like QuickThread and Fiber or POSIX needs stack or table resources for coroutine maintenance. This is a huge consumption for software running on a resource-constrained RTOS.

A. Kernel Design

We integrate protothread into the SystemC simulation kernel by a series of macro redefinitions, as described in Section II. The macros make protothread become another threading choice. SC_THREAD is implemented by protothread with flexible scheduling and synchronization methods. Other parts of the source code is kept unchanged. Our approach keeps the design as its original intention.

The macro redefinitions mainly change the structure and synchronization parts of the SystemC code. For structure, a SC_THREAD is encapsulated into a PT_THREAD by the macros defined as follows:

```
#define PTTHREAD(proc)PT_THREAD(proc(struct pt *pt)){\
    PT_BEGIN(pt); nrt ++;
#define PTEND() nrt –; PT_END(pt);}
```

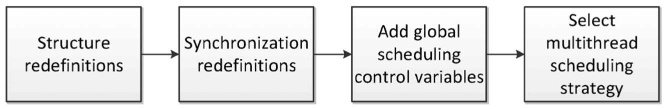
Structure redefinitions
Synchronization redefinitions
Add global scheduling control variables
Select multithread scheduling strategy

Fig. 1. Kernel design workflow.

For synchronization, the blocking synchronization in SystemC is changed to a nonblocking form. Blocking synchronization is commonly used in multithreading programming and parallel hardware execution, while it is not allowed by protothread due to its prototype design that does not have any real threads in the OS sense. Therefore, we must address the challenge of transforming all blocking synchronization statements into nonblocking ones without affecting the original semantics. We implement the nonblocking synchronization in this way: Return the execution authority to the main controller immediately if the synchronization fails, and try again in a future time that guarantees the same execution result by global scheduling. For example, the blocking read() and write() calls are redefined as follows:

```
#define sig.write(val)  PT_WAIT_UNTIL(pt,sig.nb_write(val))
#define sig.read(val)   PT_WAIT_UNTIL(pt,sig.nb_read(val))
```

Protothread also provides PT_YIELD and PT_SPAWN to implement hierarchical synchronization mechanisms. Details can be found at [3] and omitted due to space limitation.

In the kernel design, a external variable number of running threads (*nrt*) is added for the scheduler to be aware of the number of active threads, and used for control the end of the schedule sequence. Protothreads are invoked repeatedly by the scheduler to keep running, and the scheduler uses this variable to manually control the end of the invocations. When a protothread is called, we add the count by one to the variable. When a protothread quits at the last line, we reduce the count by one. The variable being zero indicates no thread is active and the software terminates. An array (*static struct pt ptt[]*) is added to store the exit points of the protothreads. It uses two bytes memory for each thread to store the entrance address of the next invocation.

Since the protothread principle is lightweight and straightforward, we can design the scheduling strategy much freely, e.g., we can use the straightforward round robin scheduling like Herrera's SWGen library [8], or event-driven scheduling like the SystemC simulation kernel, or more efficient scenario-based scheduling or quasistatic scheduling, up to your design. Basically, it is independent from the threading strategy and free of choices. The key is that protothread coroutine needs iterative invocations of all threads, no matter in which fashion, to make it an effective multithreading solution.

A SystemC model needs to be modified as follows in order to be synthesized with protothread. First, the keyword SC_THREAD is explicitly replaced using PTTHREAD, which helps find the position of a SC_THREAD declaration to perform macro redefinition. This requirement has some striking benefit for programmers can be reminded with the use of protothread. Second, thread local variables are converted to class-view global variables, i.e., class member variables, since protothread does not have a stack to store local variables. The key steps are summarized in Fig. 1.

B. Case Study

We use a small ping-pong example to illustrate code generation from SystemC, where two processes send messages back and forth in alternation. The original code is given as follows:

```
//Code for the SystemC implementation
void proc1() {                        void proc2() {
   int m=0;                              int n=0;
   while (m++<10) {                      while (n++<10) {
      FIFO2-> read();                       FIFO2-> write(n * 3);
      FIFO1-> write(n * 2);                 FIFO1-> read();
      printf("p1 ends \ n); }}              printf("p2 ends \ n); }}
```

When the program runs in SystemC simulator, *proc1* and *proc2* are two SC_THREAD processes. Thus, the synchronization of the two first-in–first-outs (FIFOs) are implemented by multithreading. The central controller needs to switch between the two threads, which makes large amount of context switching overhead. For using protothread, some extra code needs added:

```
// Code for the protothread implementation
int m=0, n=0;
PTTHREAD(proc1) {               PTTHREAD(proc2) {
    while (m++<10) {                 while (n++<10) {
      FIFO2-> read();                   FIFO2-> write(n * 3);
      FIFO1-> write(n * 2);             FIFO1-> read();
      printf("p1 ends \ n); }           printf("p2 ends \ n); }
   PTEND() }                        PTEND() }
```

And thus, our macro redefinitions act on the code and change the code to the following shape which SystemC recognizes at precompiling time:

```
//Code after precompiling for the protothreadimplementation
extern int nrt;static struct pt pt[2]; int tmp; int m=0, n=0;
PT_THREAD(proc1(struct pt*pt)) {       PT_THREAD(proc2(struct pt *pt)) {
   PT_BEGIN(pt);                          PT_BEGIN(pt);nrt++;
   while (m++<10) {                       while (n++<10) {
      PT_WAIT_UNTIL(pt,                      FIFO2-> write(n * 3);
         FIFO2-> nb_read(tmp));              PT_WAIT_UNTIL(pt,
      FIFO1-> write(n * 2);                    FIFO1- >nb_read(tmp));
      printf("p1 ends \ n); }                printf("p2 ends \ n); }
   nrt--;PT_END(pt); }                    nrt--; PT_END(pt); }
```

With the "PT_WAIT_UNTIL()" macro, the original blocking synchronization of read() in a thread is implemented in a nonblocking way, i.e., it returns back to the central controller by "PT_WAIT_UNTIL()" if no data is obtained at the time of its execution, e.g., *proc1*. The central controller may let the other protothread (*proc2*) execute and generate a token, and return back for no data to read. The central controller will invoke the first thread again and at this time a token can be read and the thread may continue to run, and so on. Protothread actually constructs a single execution sequence with no blocking from the original multithread. The system time used for thread context switch changes to zero, and the system memory space used is largely reduced.

IV. Related Work and Discussions

Coroutines in the SystemC simulation kernel are OS dependent: QuickThread needs UNIX instructions, Fiber needs Win32 APIs, and POSIX needs OS support [1]. Compared to that, our approach needs no support from OS to maintain multithreading, and only takes 2-byte memory overhead for one thread. Experimental results show that many kB memory is occupied for tens of threads maintenance. The SystemC simulation kernel uses an event-driven method to schedule all threads. Even if only two modules are doing context switching between each other and leave all other modules free, the kernel still explores the whole space of all modules because all threads try to get running at the system level though only two threads really need to. Our implementation strides over this problem for protothread will not try preemption automatically. Event-driven scheduling, quasistatic scheduling and the simple round robin scheduling are all applicable in our approach. Users can get rid of the limitation of event-driven scheduling of the SystemC simulation kernel, and select any scheduling strategy which is more efficient for the application. Our technique is more memory friendly to lightweight RTOS, and thus is more flexible for embedded applications.

SWGen [8] is a library subset abstracted from SystemC source code that targets embedded software generation. SWGen library uses POSIX thread to implement SC_THREAD, which is an OS-independent implementation to a certain extent since most RTOS supports POSIX. POSIX is a system-level thread, which needs a large piece of memory space to maintain correct execution, and can produce much overhead when context switching is conducted by OS kernel. Our approach relies on protothread, which is a lightweight, memory-free and fast coroutine, and can provide improved compatibility and efficiency by minimizing the context switching overhead and avoiding useless system-level preemption attempts from free POSIX threads.

Sharad *et al.* [6] presented an effective technique to improve the simulation efficiency of SystemC designs. Context switching is managed manually by adding finite state controllers to the SystemC threads, and system overhead is minimized. Our technique targets efficient software synthesis, and supports some additional features including hierarchical threading, flexible multithread scheduling, lower memory overhead, and improved applicability and extensibility due to the systematic coroutine implementation.

V. Performance Evaluation

We conducted a series of performance evaluation experiments to compare our approach with the software synthesis technique in SWGen [8] and the SystemC simulation kernel [1]. The SystemC models of three realistic applications, flip-flop switcher, MPEG decoder, and GSM communication module are used for code generation. We execute these programs for 50 000 and 500 000 clock cycles, respectively, and obtain their performance on time and memory usage. The experiments are run on a workstation with a 400 MHz CPU and 512 MB RAM. Table IV gives the performance evaluation results. The flip-flop switching example shows the most significant performance improvement, since it does few things between thread switching. The average performance of our approach for common embedded applications such as MPEG and GSM is of 31.9% and 65.5% reductions in user-level execution times compared to SWGen and SystemC, and zero system-level time consumptions compared to significant system time consumed by the other techniques. The memory space is reduced by 87.0% and 67.9%, on average.

TABLE IV
COMPARISON OF OUR APPROACH WITH SWGEN [8]

Application		Flip-flop		MPEG		GSM	
No. cycles		50K	500K	50K	500K	50K	500K
User	Ours	0.04	0.33	3.5	34.79	5.06	72.9
time	SWGen	0.13	1.22	4.69	47.23	11.76	89.75
(s)	SystemC	0.71	7.08	53.8	537.4	9.87	98.59
Sys.	Ours	0	0	0	0	0	0
time	SWGen	0.44	6.49	1.81	18.45	7.22	87.83
(s)	SystemC	0.02	0.03	0.04	0.03	0.02	0.04
Total	Ours	0.11	0.42	3.54	34.84	9.15	73.22
time	SWGen	0.62	7.8	5.51	57.47	17.06	179.65
(s)	SystemC	0.78	7.28	54.03	538.1	10.03	98.81
Memory	Ours	2.58	8.3	8.31	8.31	8.62	8.62
(MB)	SWGen	24.7	24.7	65.7	65.7	197.1	197.1
	SystemC	18.6	18.6	22.1	22.1	32.48	32.48

VI. Conclusion

We have presented a software synthesis approach for SystemC designs based on coroutine techniques to reduce the execution overheads of the generated software, which is more efficient than the conventional OS threads based approaches. The synthesized software is platform/RTOS independent, stackless threading with greatly reduced context switching overheads on time and memory usage, and does not have limitations on the selection of central scheduling strategies. For an embedded system with many threads, the performance of the software synthesized using our approach is significantly better than that by the conventional approaches, as verified by the experiments.

References

[1] SystemC [Online]. Available: www.systemc.org

[2] J. Xu and W. Wolf, "Platform-based design and the first generation dilemma," in *Proc. Electron. Design Process Workshop (EDP)*, Monterey, CA, Apr. 2002.

[3] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying event-driven programming of memory-constrained embedded systems," in *Proc. 4th Int. Conf. Embed. Netw. Sensor Syst.*, New York, 2006, pp. 29–42.

[4] *IEEE Standard for Information Technology- Portable Operating System Interface (Posix) Base Specifications, Issue 7*, IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004), Dec. 2008, pp. c1 -3826.

[5] GNU Compiler [Online]. Available: http://gcc.gnu.org

[6] S. A. Sharad and S. K. Shukla, "Optimizing system models for simulation efficiency," *Formal Method. Model. Syst. Design: Syst. Level Perspect.*, pp. 317–330, 2004.

[7] R. Buchmann and A. Greiner, "A fully static scheduling approach for fast cycle accurate systemc simulation of mpsocs," in *Proc. Int. Conf. Microelectron. ICM 2007*, Cairo, Egypt, Dec. 2007, pp. 101–104.

[8] F. Herrera, H. Posadas, P. Sanchez, and E. Villar, "Systemetic embedded software generation from systemc," in *Proc. Conf. Design, Autom., Test Eur. (DATE '03)*, Washington, DC, 2003, p. 10142, IEEE Computer Society.