Chapter 17
Odds & Ends

Performance, Gotchas, and Tidbits

This chapter wraps up with a few simple observations on using SystemC to its greatest advantage. The authors provide hints about ways to keep simulation performance high and provide observations about the modeling language in general. This chapter contains no exercises. We leave application to your individual creativity.

17.1 Determinants in Simulation Performance

We sometimes hear comments from folks such as, "We tried SystemC, but our simulations were slower than Verilog." Such comments betray a common misconception. SystemC is not a faster simulator. The OSCI reference version of the SystemC simulator has several opportunities for optimization, and there are EDA vendors hoping to capitalize on that situation. More importantly, simulation performance is not so much about the simulator as it is the way the system is modeled.

KEY POINT: SystemC simulation speed is linked directly to the use of higher levels of modeling using un-timed and transaction-level concepts.

For all simulators (e.g., SPICE, Verilog, VHDL, or SystemC), there are a fundamental set of tasks that must be performed: moving data, updating event queues, keeping track of time, etc. Any simulator simulating detailed pin-level activity and timing information will provide a certain level of performance. Almost all simulators for a given class of detail will perform within a factor of two or so.

No so long ago, cycle-based simulators were all the rage due to their advertised speed. Problems arose when designers discovered that these simulators didn't provide the same level of accuracy as their event-driven counterparts. Indeed, that was exactly the reason they ran faster!

That said, RTL simulates at RTL speeds. Certainly, there are simulators that do RTL better than others, but they still have the limitation of keeping track of all the same details. A good optimizer may improve performance by finding commonalities, but the improvement will be bounded.

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_17, © Springer Science+Business Media, LLC 2010

GUIDELINE: To improve simulation performance, reduce details and model at higher levels of abstraction whenever possible.

It is possible to obtain dramatic speed improvements by keeping as much of the system as possible at very high levels of abstraction, and only using details where absolutely required. This approach has the effect of minimizing the number of simulation context switches, which will keep performance high. That said, even if you are at the right level of abstraction, it is necessary to limit the number of calls to `wait()`.

Part of the problem lies with understanding what a given simulation is supposed to accomplish. Ask yourself, "What question is this simulation model supposed to answer?" For example, early in the design process the architect may wish to know if a new algorithm even works. At this level, timing and pins are not really interesting. A simple executable that takes input data and produces output for analysis is all that is required. Timing should not be a part of this model. Sequential execution is probably sufficient.

Another set of questions might be, "Have all the parts been connected? Have we defined paths for all the information required to perform the system functions?" These questions may be answered by creating a module for every component and using a simple transaction-level model to interconnect the pieces. Cycle accuracy should not be a concern at this point in the design.

17.1.1 Saving Time and Clocks

How can you live without time or clocks[1]? This is really quite simple. For instance, suppose you need to model time to determine performance. Rather than coding a wait for N rising edges, it is much more efficient to simply delay by *N**`clock_period`.

Another common technique occurs when using handshakes. If you need to wait for a signal, then simply wait on the signal directly. The hardware may do sampling at clock edges, but that wastes time. If you really need to synchronize to the clock, then do both, as follows:

```
wait(acknowledge->posedge_event());
if (not clock->event()) wait(clock->posedge_event());
```

Fig. 17.1 Synchronized wait for a signal

Perhaps you need to transfer information from one port to another in the design. Even though you know the result will be delayed through a FIFO over multiple clocks, there is no need to create a FIFO. Just read it from the input, delay, and write it to the output.

[1] We ask this question from an electronic system design perspective, not from a philosophical perspective.

```
In->read(packet);
wait(50*clock_period);
Out->write(packet);
```

Fig. 17.2 Example of FIFO elimination

Events are also a powerful way of communicating information. If you don't really need to test the value of a signal but are only interested in the change, it is more efficient to use an event than an **`sc_signal<bool>`**. Earlier in the book, we illustrated some primitive channels to do just this (e.g., in the `heartbeat` example).

Does your clock really need to oscillate at 100 GHz? Perhaps it would suffice to use a higher level clock. Do you really need GHz, or are MHz or even KHz sufficient?

Another overlooked issue is using too much or too little resolution. For instance, resolution should probably allow distinguishing at least clock periods for the fastest modeled clock.

17.1.2 Moving Large Amounts of Data

So, the model is efficiently using time, but it still seems to be simulating too slowly. Perhaps you are attempting to move too much data. Do you really need to move the data or do you just need to record the fact that data was moved and that an appropriate amount of time has passed?

For example, perhaps you could model the movement of a chunk of data as follows instead of moving the actual data:

```
struct payload {
unsigned long byte_count;
unsigned value; // a single unique value
};
```

Fig. 17.3 **`struct`** for `payload`

Now, you will need to modify the read/write routines in the channels to do something like this:

```
void Bus<payload>::write(unsigned addr, payload data)
{
  wait(data.byte_count*t_BYTE_DELAY);
  // transfer the data
}
```

Fig. 17.4 Bus write with `payload`

Perhaps, you need to transfer the data, but how much data do you really need to test the problem at hand? For instance, if dealing with video graphics, would a small 64 x 48 pixel buffer suffice to test an algorithm, rather than a full 640 x 480 or larger frame?

Perhaps, you need to transfer a large block of data across the bus, but can you model it using smart pointers instead? In other words, manage the chunk of simulator memory with a pointer. We recommend you use a Boost.org smart pointer, or the equivalent, to avoid problems with memory leaks or corruption.

Thus, you might have:

```
struct payload {
  unsignedlong byte_count;
  smart_ptr<int> pValues;
  payload(unsignedlong bc)
  : byte_count(bc)
  {
    pValues = new int[bc];
  }
};
```

Fig. 17.5 Smart pointer with `payload`

Now, you are simply passing around a pointer and only manipulating the data when it really needs to be manipulated.

Do you really need to fully populate a memory, or would a sparse memory model suffice? The SystemC Verification library contains a very nice sparse memory model that is very easy to use.

17.1.3 Too Many Channels

Another interesting area for SystemC designers to watch is channels. Every channel interaction involves at least two calls (producer and consumer), two events, and possibly two copy operations. Hierarchical **`sc_port`** to **`sc_port`**connections are very efficient because they simply pass a pointer to the target channel at elaboration. You will also find **`sc_export`** to **`sc_export`** hierarchical connections are similarly efficient. Another efficient way of communicating is **`sc_port`** to **`sc_export`**, since it can be implemented as a simple function call. If you find yourself writing a process that merely copies one port to another, consider the possibility of re-architecting the connectivity.

17.1.4 Effects of Over Specification

Often designers tend to think in terms of the final implementation rather than the general problem being designed. This approach sometimes results in too much specification. For instance, a behavior may be specified as a finite state machine (FSM), when the real issue is simply a handshake or data transfer. Be careful when presented with myriads of detail to abstract the real needs of the design.

17.1.5 Keep it Native

Keeping data native has already been discussed under data types earlier in the book, but this topic bears repeating. Data types are an abused subject. Does the model at hand really need to specify 17 bits, or would a simple **`int`** suffice? Native C++ data types will simulate many times faster than their SystemC hardware-specific counterparts. Similarly, what do you gain using **`sc_logic`**? Is the unknown value relevant to the current level of modeling? Once again, the issue is to model only those items that will affect the results of the simulation.

17.1.6 C++ Compiler Optimizations

Depending on the stability of your model, you may want to consider looking at optimizing your use of the C++ compiler. Many times, default `make` scripts assume that the developer wants maximum debug visibility, and the compiler obliges with additional visibility that may affect simulation performance.

When looking for maximum performance, make sure that your SystemC library and your system design are compiled without a debug option. Additionally, some compilers have switches that perform additional run-time optimizations at the expense of increased compile time. If you plan to run extensive simulation with the same model, it may pay to wade through the documentation for your compiler.

Another example, ensuring that **`#ifndef`** is on the first line of a header file improves performance for some compilers.

17.1.7 C++ Compilers

Many folks begin their SystemC explorations on the native C++ compiler that comes with their system, usually either GNU g++ or Microsoft Visual C++. It should not be a surprise that commercial alternatives exist with even better optimization.

17.1.8 Better Libraries

The STL library that comes with most C++ compilers is not always the most efficient implementation. There are commercial implementations of the STL that

should have much higher performance. The same can be said for the SystemC libraries that come from OSCI. If you are concerned with performance, it is probably worth your time to investigate your options with commercial solutions if you can afford it.

17.1.9 Better and More Simulation Computers

At the current rate of improvement in cost-performance, be sure you are running on the latest technology. It's a shame to not be spending $300-$1,000 for a potential 2x-3x performance gain. Similarly, why not increase the number of compute engines and run two, three, ten, or even hundreds of simulations at the same time. Compute farms are very effective for many types of modeling problems.

17.2 Features of the SystemC Landscape

Because SystemC is a C++ class library rather than a truly independent language, SystemC has some aspects that seem to annoy its users (particularly experienced designers from an RTL background). This section simply notes these aspects. Keep in mind that part of the power of SystemC is the fact that it is C++, and therefore, it is extremely compatible with application software.

17.2.1 Things You Wish Would Just Go Away

For the novice, just getting a design to compile can be a challenge. This section lists some of the most common problems. All of them relate directly to C++.

Syntax errors in **#include** files often are reported as errors in the including implementation (i.e., .cpp file). The most common error is forgetting to put the trailing semicolon on a **SC_MODULE**, which is really a **class**.

The use of **semicolons** in C++ may seem odd at times. The **class** and **struct** require a closing semicolon.

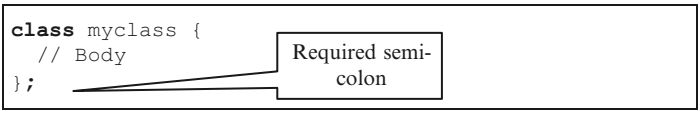

Fig. 17.6 C++ class requires semicolon

On the other hand, function definitions and code blocks do not require a semicolon.

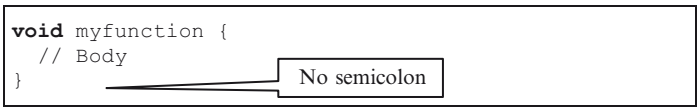

Fig. 17.7 C++ function does not use semicolon

Similarly, **SC_FORK**/SC_**JOIN**have the odd convention of using commas. This punctuation is used because they are really just fancy macros.

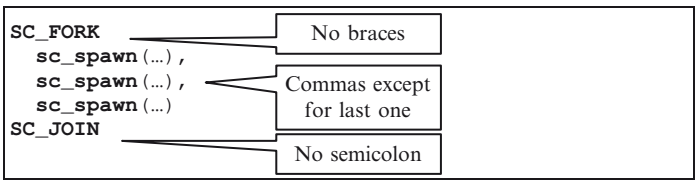

Fig. 17.8 C++ fork/join idiosyncrasy

SystemC relies heavily on templates. The templates have the annoying space between the greater than brackets.

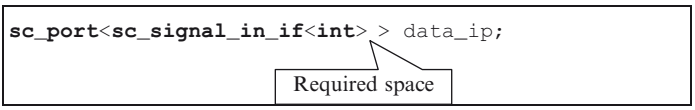

Fig. 17.9 C++ template idiosyncrasy

Inside the basic syntax of a module, **sensitive** and **dont_initialize** methods must be tied to the immediately preceding **SC_THREAD**, **SC_METHOD** or **SC_CTHREAD** registration. This tying is usually a lot easier to deal with if you indent the code slightly relative to the registration. For example:

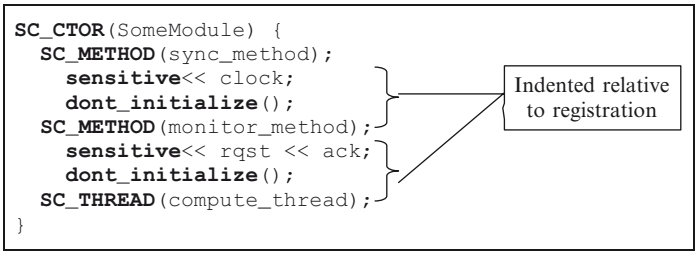

Fig. 17.10 Example of using indents to highlight registrations

dont_initialize brings up another issue common to **SC_METHOD**. Unless you specify otherwise, all processes are executed once at initialization despite the appearance of static sensitivity unless **dont_initialize** is used. For some, this behavior can be confusing at first. Try to remember that all simulation processes are run during initialization unless **dont_initialize** is applied.

17.2.2 Development Environment

What is the best way to address these idiosyncrasies besides just learning them? We highly recommend obtaining language-sensitive text editors with color highlighting, and we recommend obtaining lint tools designed specifically for SystemC. The authors' favorite text editor is vim in graphical mode, also known as gvim. You can obtain a copy of vim from www.vim.org for almost any platform.

Other users are quite successful using emacs (graphical of course) or nedit. All three of these editors have environments available for download that support SystemC. You can obtain these from our web site.

Another interesting environment is Eclipse.

There are a few C++ lint tools and at least one lint tool focused on SystemC that is commercially available[2]. Some EDA tools have built-in lint-like checkers. Your mileage will vary, and we highly recommend a careful evaluation before committing to any of these tools.

17.2.3 Conventions and Coding Style

Coding styles are a well-known issue, and lots of C++ rules and guidelines exist. Since SystemC is C++, this is a good starting point for SystemC coding guidelines.

Hardware designers have special issues to consider. Probably one of the best books written on this subject for hardware design is the *Reuse Methodology Manual for System-on-a-Chip Design*s by Michael Keating and Pierre Bricaud. Most of the concepts presented there have direct application to SystemC. Let's just touch on a few.

A name is a name, right? Wrong! Names of classes, variables, functions, and other matters are a critical part of making your code readable and understandable. If you have been observant, you will notice we've inserted various naming conventions specific to SystemC in the examples. For instance, processes always have a suffix of `_thread`, `_method` or `_cthread`. This convention is used because **wait**`()` results in a run-time error when used with **SC_METHOD**, and visa versa for **next_trigger**`()`.

[2] Actis Design www.actisdesign.com.

Similarly, we adopted a convention when addressing ports and probably you should do likewise for using anything **sc_signal**<*T*> or otherwise supporting the evaluate-update paradigm.

17.3 Next Steps

If you have read this far, you are probably considering adopting SystemC for an upcoming project. Or, perhaps you have already started, and you are looking for help moving forward. This section provides some ideas.

17.3.1 Guidelines for Adopting SystemC

In the fall of 2003, the authors presented a paper on the subject of language adoption, "How to Really Mess Up Your Project Using a New Language" at the Synopsys User's Group in Boston, MA. We included a number of key points, which we provide for your consideration.

1. Don't do it alone—Obtain management support.
2. Doing things the same way will produce the same results regardless of the language.
3. Look at the big picture, the product or system—Not the small tasks.
4. Don't skimp on training—Obtain good formal training.
5. Obtain mentoring.
6. Adopt the new paradigm to gain the advantages of a new language.
7. Specifications should use the appropriate level of abstraction for the new paradigm.
8. Put coding discipline in place quickly with coding guidelines, lint tools, and reviews.
9. Choose templates approved by seasoned experts in the new language.
10. Start automation and environment simply and cleanly.
11. Evaluate EDA tools for the big picture.
12. Insist on well-documented and supported tools in all areas including tools version and configuration.
13. Apply the technology to a pilot project that focuses on the big picture.

There are a number of companies supporting SystemC methodologies. A quick visit to the OSCI web site www.systemc.org can provide a starting point. Or, visit our web site, www.scftgu.com, for our view.

17.3.2 Resources for Learning More

For the readers who would like more information on SystemC extensions, we recommend the following resources for further study.

Table 17.1 SystemC resources

	Type	Details
1	Web site	Starting point for SystemC. Retrieved March 2004 from: http://www.systemc.org/. This site has several great papers and white papers as well as email forums for getting help or discussing SystemC.
2	Web site	The European SystemC Users Group web site. This site has additional quality papers and additional news and activities. Retrieved March 2004 from: http://www-ti.informatik.uni-tuebingen.de/~systemc/
3	Web site	The web site for the North American SystemC Users Group. Focused on SystemC activities in North America. Retrieved March 2004 from: http://www.nascug.org/
4	Web site	References for SystemPerl HDL tools. Retrieved March 2004 from: http://www.veripool.com/
5	Web site	The web site for sharing Open Source SystemC IP, tool and concepts. Retrieved July 2007 from http://www.greensocs.org/
6	Web site	The web site for the Latin America SystemC Users Group. Focused on SystemC activities in South America. Retrieved August 2007 from http://www.lascug.org/

For the readers still gasping for help with C++, here are some additional recommendations for further study.

Table 17.2 C++ resources

	Type	Details
1	Book	Koenig, A., Moo, B. *Accelerated C++*. Boston: Addison-Wesley, 2000. A highly recommended textbook for learning to speak C++ natively.
2	Book	Stroustrup, B. *The C++ Programming Language*. Florham Park, New Jersey:Addison_Wesley, 2000. Probably the best C++ reference and is written by the creator of C++.
3	Book	Loudon, K. *C++ Pocket Reference*. Sebastopol, California: O'Reilly & Associates, Inc., 2003. A convenient and reasonably organized quick reference. Good for those who are not yet C++ experts.
4	Book	Josuttis, N. *The C++ Standard Library*: A Tutorial and Reference. Indianapolis, Indiana: Addison-Wesley, 1999. A complete reference and good tutorial for the STL, a very useful library for modeling.
5	Web site	Stroustrup, B. Definitive reference for C++ by the author of C++. Retrieved March 2004 from:http://www.research.att.com/~bs/C++.html
6	Tool	van Heesch, D. Documentation system for C++ code. Retrieved March 2004 from: http://www.stack.nl/~dimitri/doxygen/index.html
7	Web site	References for C++ programming. Retrieved March 2004 from: http://www.cplusplus.com/
8	Book	Henricson, M., Nyquist. E. Industrial Strength C++. Upper Saddle River, New Jersey: Prentice Hall, 1996. (Online Version: http://www.elho.net/dev/doc/industrial-strength.pdf)
9	Web book	A free online book. Retrieved March 2004 from: http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html
10	Article	Hoff, T. C++ Coding Standard. Retrieved March 2004 from: http://oopweb.com/CPP/Documents/CodeStandard/VolumeFrames.html

(continued)

Table 17.2 (continued)

	Type	Details
11	Article	Baldwin, J. 1992. An Abbreviated C++ Code Inspection Checklist. Retrieved March 2004 from: http://www.chris-lott.org/resources/cstyle/Baldwin-inspect.pdf
12	Book	Pressman, R. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2001. (Online Version: http://www.rspa.com/about/sepa.htm)
13	Web class	Free C++ class based on an inexpensive tool. Retrieved March 2004 from: http://www.codeWarriorU.com/
14	Web class	Free C++ class. Retrieved March 2004 from:http://www.free-ed.net/fr03/lfc/030203/120/
15	Book	Sutter,H & Alexandrescu, A. *C++ Coding Standards*, Addison-Wesley, 2004
16	Book	McConnell, S. Code Complete, 2nd Ed., Microsoft Press, 2004

We hope you'll be ready for our next book when we introduce topics such as the SystemC assertions, and we go deeper into a discussion of SystemC design methodologies and design styles. We also expect to provide updates as SystemC version 2.3, which is just now appearing on the horizon and will effect changes to the standard.