An Efficient C++ Framework for Cycle-Based Simulation

J.P. Grossman

Abstract

*System design usually begins with a high-level software simulation which is later refined to a detailed hardware description. A simulation framework can facilitate this process by automating certain hardware abstractions, providing important debugging support, and allowing the transition from a high-level simulation to a low-level hardware description to occur within a single code base. This paper discusses our experience with **Sim**, a C++ framework developed as an internal tool for cycle-based simulations. We describe the features of Sim which were found to be most helpful to the design process, and we compare Sim to a publicly available simulation framework. The Sim framework is efficient; typical low-level hardware simulations run with a slowdown of less than 2x compared to straight C++ implementations.*

1 Introduction

Software simulation is a critical step in the hardware design process. Hardware description languages such as Verilog and VHDL allow designers to accurately model and test the target hardware, and provide a design path from simulation to fabrication. However, they are also notoriously slow, and as such are not ideal for simulating long runs of a large, complex system. Instead, a high-level language (usually C or C++) is used for initial functional simulations. Inevitably, the transition from this high-level simulation to a low-level hardware description language is a source of errors and increased design time.

Recently there have been a number of efforts to develop simulation frameworks that enable the accurate description of hardware systems using existing or modified general-purpose languages ([Ku90], [Liao97], [Gajski00], [Ramanathan00], [Cyn01], [SC01]). This bridges the gap between high-level and register transfer level simulations, allowing designers to progressively refine various components within a single code base. The approach has been successful: one group reported using such a framework to design a 100 million transistor 3D graphics processor from start to finish in two months [Kogel01].

Project Aries Technical Memo ARIES-TM-17
Artificial Intelligence Laboratory
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA

Sponsored by DARPA/AFOSR Contract Number F306029810172

There are four important criteria to consider when choosing or developing a framework:

Speed: The simulator must be fast. Complex simulations can take hours or days; a faster simulator translates directly into reduced design time.

Modularity: There should be a clean separation and a well-defined interface between the various components of the system.

Ease of Use: The framework should not be a burden to the programmer. The programming interface should be intuitive, and the framework should be transparent wherever possible.

Debugging: The framework must contain mechanisms to aid the programmer in detecting errors within the component hierarchy.

These criteria were used to create **Sim**, a cycle-based C++ simulation framework developed as an internal tool for the purpose of modeling a large parallel machine. Through experience we found that Sim met all four criteria with a great deal of success. In this paper we describe the Sim framework and we report on what we observed to be its most useful features.

The following section describes the Sim framework and the debugging support that it provides. In Section 3 we measure the performance of simulations running under Sim compared to simluations coded in straight C++. In Section 4 we contrast Sim with SystemC [SC01], an open-source C++ simulation framework supported by a number of companies. We offer our conclusions in Section 5.

2 The Sim Framework

To a large extent, the goals of speed and modularity can be met simply by choosing an efficient object-oriented language, i.e. C++. What distinguishes a framework is its simulation model, programming interface and debugging features. Sim implements a pure cycle-based model; time is measured in clock ticks, and the entire system exists within a single clock domain. The following sections describe the programming concepts and debugging mechanisms of the Sim framework.

2.1 Overview

Sim provides the programmer with three abstractions: components, nodes and registers. A component is a C++ class which is used to model a hardware component. In debug builds, Sim automatically generates hierarchical names for the components so that error messages can give the precise location of faults in the simulated hardware. A node is a container for a value which supports connections and, in debug builds, timestamping. Nodes are used for all component inputs and outputs. Registers are essentially D flip-flops. They contain two nodes, D and Q; on the rising clock edge D is copied to Q.

Simulation proceeds in three phases. In the construction phase, all components are constructed and all connections between inputs and outputs are established. When an input/output node in one component is connected to an input/output node in another component, the two nodes become synonyms, and writes to one are immediately visible to reads from the other. In the initialization phase, Sim prepares internal state for the simulation and initial values may be assigned to component outputs. Finally, the simulation phase consists of alternating calls to the top-level component's Update function (to simulate combinatorial evaluation) and a global Tick function (which simulates a rising clock edge).

Figure 1 gives an example of a simple piece of hardware that computes Fibonacci numbers and its equivalent description using Sim. The example shows how components can contain sub-components (Fibonacci contains a ClockedAdder), how nodes and registers are connected during the construction phase (using the overloaded left shift operator), and how the simulation is run at the top level via alternating calls to fib.Update() and Sim::Tick(). The component functions Construct() and Init() are called automatically by the framework; Update() is called explicitly by the programmer.

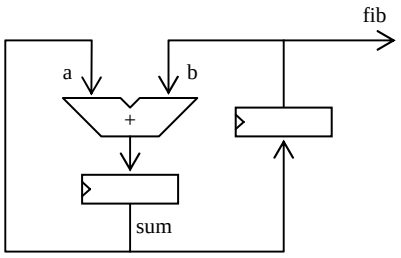
2.2 Timestamps

In a cycle-based simulator, there are three common sources of error:

Invalid Outputs: The update routine(s) for a component may neglect to set one or more outputs, resulting in garbage or stale values being propagated to other components.

Missing Connections: One or more of a component's inputs may never be set.

Bad Update Order: When the simulation involves components with combinational paths from inputs to one or more outputs, the order in which components are updated becomes important. An incorrect order-




```
class ClockedAdder : public CComponent
{
    DECLARE_COMPONENT(ClockedAdder)
public:
    Input<int>  a;
    Input<int>  b;
    Output<int> sum;

    Register<int> reg;

    void Construct (void) {sum << reg;}
    void Init (void) {reg.Init(0);}
    void Update (void) {reg = a + b;}
};

class Fibonacci : public CComponent
{
    DECLARE_COMPONENT(Fibonacci)
public:
    Output<int> fib;

    Register<int> reg;
    ClockedAdder  adder;

    void Construct (void) {
        adder.a << adder.sum;
        adder.b << reg;
        reg << adder.sum;
        fib << reg;
    }
    void Init (void) {
        adder.sum.Init(1);
        reg.Init(0);
    }
    void Update (void) {adder.Update();}
};

void main (void)
{
    Fibonacci fib; // Construction
    Sim::Init();   // Initialization
    while (1) {    // Simulation
        fib.Update();
        Sim::Tick();
    }
}
```

Figure 1: Schematic and Sim code for Fibonacci

```
Warning: Fibonacci0::Adder0::Input1
    Invalid timestamp
    c:\projects\aries\sim\sim.h, line 527,
        simTime = 1
```

Figure 2: Warning message generated if the programmer forgets the connection "adder.b << reg"

ing can have the effect of adding or deleting registers at various locations.

While careful coding can avoid these errors in many cases, experience has shown that it is generally impossible to write a large piece of software without introducing bugs. In addition, these errors are particularly difficult to track down as in most cases they produce silent failures which go unnoticed until some unrelated output value is observed to be incorrect. The programmer is often required to spend enormous amounts of time finding the exact location and nature of the problem.

The Sim framework helps the programmer to eliminate all three sources of error by timestamping inputs and outputs. In debug builds, each time a node is written to it is timestamped, and each time a node is read the timestamp is checked to ensure that the node contains valid data. When an invalid timestamp is encountered, a warning message is printed which includes the automatically generated name of the input/output, pinpointing the error within the component hierarchy.

Timestamped nodes have proven to be by far the most useful feature of the Sim framework. They can speed up the debugging process by an order of magnitude, allowing the programmer to detect and correct errors in minutes that would otherwise require hours of tedious work. Figure 2 shows the exact warning message that would be generated if the connection "adder.b << reg" were omitted from the function Fibonacci::Construct() in Figure 1.

2.3 Other Debugging Features

The Sim framework provides a number of Assert macros which generate warnings and errors. As is the case with standard assert macros, they give the file and line number at which the error condition was detected. In addition, the error message contains the simulation time and a precise location within the component hierarchy (as shown in Figure 2). Again, this allows the programmer to quickly determine which instance of a given component was the source of the error.

When one node A is connected to another node B, the intention is usually to read from A and write to B (note that order is important; connecting A to B is not the same as connecting B to A). Timestamps can be used to detect reads from B, but not writes to A. To detect this type of error, in debug builds a node connected to another node is marked as read-only[1]; assignment to a read-only node generates a warning message. In practice this feature did not turn out to be very useful as the simple naming convention of prefixing inputs with in_ and outputs with out_ tended to prevent these errors. The feature does, however, provide a safety net, and it does not affect release builds of the simulator.

3 Performance Evaluation

Making use of a simulation framework comes at a cost, both in terms of execution time and memory requirements. In this section we will attempt to quantify these costs for the Sim framework by implementing four low-level benchmark circuits in both Sim and straight C++. The most important difference between the implementations is that inputs and outputs in the C++ versions are ordinary class member variables; data is propagated between components by explicitly copying outputs to inputs each cycle according to the connections in the hardware being modeled.

3.1 Benchmark Circuits

The following are the benchmark circuits used in our evaluation:

LFSR: 4-tap 128-bit linear feedback shift register. Simulated for 2^{24} cycles.

LRU: 1024 node systolic array used to keep track of least recently used information for a fully associative cache. Simulated for 2^{17} cycles.

NET: 32x32 2D grid network with wormhole routing. Simulated for 2^{13} cycles.

FPGA: 12 bit pipelined population count implemented on a simple FPGA. The FPGA contains 64 logic blocks in an 8x8 array; each block consists of a 4-LUT and a D flip-flop. Simulated for 2^{21} cycles.

In the Sim version of FPGA, the FPGA configuration is read from a file during the construction phase and used to make the appropriate node connections. In the C++ version, which does not have the advantage of being able to directly connect inputs and outputs, the configuration is used on every cycle to manually route data between logic blocks.

3.2 Results

The benchmarks were compiled in both debug and release modes and run on a 1.2GHz Pentium III processor. Table 1 shows the resulting execution times in seconds, and Table 2 lists the memory requirements in bytes.

[1] Unless the node has been declared as a bi-directional Input/Output.

	Debug			Release		
	C++	Sim	Ratio	C++	Sim	Ratio
LFSR	17.56	500.40	**28.50**	5.77	20.56	**3.56**
LRU	15.78	106.54	**6.75**	3.15	5.14	**1.63**
NET	11.34	126.54	**11.16**	2.86	5.38	**1.88**
FPGA	12.29	6.42	**0.52**	3.44	0.36	**0.10**

Table 1: Benchmark execution time in sectonds

	Debug			Release		
	C++	Sim	Ratio	C++	Sim	Ratio
LFSR	129	7229	**56.04**	129	1673	**12.97**
LRU	28672	233484	**8.14**	28672	61448	**2.14**
NET	118784	806936	**6.79**	118784	249860	**2.10**
FPGA	9396	74656	**7.95**	7084	14598	**2.06**

Table 2: Benchmark memory requirements in bytes

The time and space overheads of the Sim framework are largest for the LFSR benchmark; the release build runs 3.56 times slower and requires 12.97 times more memory than the corresponding C++ version. This is because the C++ version is implemented as a 128-element byte array which is manually shifted, whereas the Sim version is implemented using 128 actual registers which are chained together. In release builds, each register contains three pointers: one for the input (D) node, one for the output (Q) node, and one to maintain a linked list of registers so that they can be automatically updated by the framework when Sim::Tick() is called. This, together with the 129 bytes of storage required for the actual node values, accounts for the factor of 13 increase in memory usage. Clearly the register abstraction, while more faithful to the hardware being modeled, is a source of inefficiency when used excessively.

The execution time and memory requirements for the release builds of the other three Sim benchmarks compare more favorably to their plain C++ counterparts. In all cases the memory requirements are roughly doubled, and the worst slowdown is by a factor of 1.88 in NET. In the FPGA benchmark the Sim implementation is actually faster by an order of magnitude. This is due to the fact that the framework is able to directly connect nodes at construction time as directed by the configuration file.

Not surprisingly, the Sim framework overhead in the debug builds is quite significant. The debug versions run roughly 20-25 times slower than their release counterparts, and require four times as much memory. This is largely a result of the node time-stamping that is implemented in debug builds.

4 Comparison with SystemC

SystemC is an open source C++ simulation framework originally developed by Synopsys, CoWare and Frontier Design. It has received significant industry support; in September 1999 the Open SystemC Initiative was endorsed by over 55 system, semiconductor, IP, embedded software and EDA companies [Arnout00].

The most important difference between Sim and SystemC is that, like Verilog and VHDL, SystemC is event driven. This means that instead of being called once on every cycle, component update functions are activated as a result of changing input signals. Event driven simulators are strictly more powerful than cycle-based simulators; they can be used to model asynchronous designs or systems with multiple clock domains.

Event driven simulation does, of course, come at a price. A minor cost is the increased programmer effort required to register all update methods and specify their sensitivities (i.e. which inputs will trigger execution). More significant are the large speed and memory overheads of an event driven simulation kernel. For example, we implemented the LRU benchmark using SystemC, and found that the release version was over 36 times slower and required more than 8 times as much memory as the Sim release build.

While event driven simulation is more powerful

```
class BlackBox : public CComponent
{
    DECLARE_COMPONENT(BlackBox)
public:
    Input<int>  a;
    Input<int>  b;
    Output<int> out;

    void Update (void);
};

class GreyBox : public CComponent
{
    DECLARE_COMPONENT(GreyBox)
public:
    Input<int>  in;
    Output<int> out;

    BlackBox   m_box;
    int        m_key;

    void Update (void) {
        m_box.a = in;
        m_box.b = m_key;
        m_box.Update();
        out = in + m_box.out;
    }
};
```

Figure 3: Inlining a combinational component (BlackBox) within the GreyBox Update() function.

in terms of the hardware it can simulate, it also presents a more restrictive programming model. In particular, the programmer has no control over when component update functions are called. Hiding this functionality ensures that updates always occur when needed and in the correct order, but it also prevents certain techniques such as inlining a combinational component within an update function. Figure 3 gives an example of such inlining in the Sim framework; it is simply not possible using SystemC.

Another important difference between Sim and SystemC is the manner in which inputs and outpus are connected. Sim allows input/output nodes to be directly connected; in release builds a node is simply a pointer, and connected nodes point to the same piece of data. SystemC, by contrast, requires that inputs and outputs be connected by explicitly declared signals. This approach is familiar to hardware designers from hardware description languages such as Verilog and VHDL. However, it is less efficient in terms of programmer effort (more work is required to define connections between components), memory requirements, and execution time.

A minor difference between the frameworks is that SystemC requires component names to be supplied as part of the constructor, whereas Sim generates them automatically. In particular, components in SystemC have no default constructor, so one cannot create arrays of components in the straightforward manner, nor can components be member variables of other components. The programmer must explicitly create all components at run time using the *new* operator. Clearly this is not a fundamental difference and it would be easy to fix in future versions of SystemC. It does, however, illustrate that the details of a framework's implementation can noticeably affect the amount of programmer effort that is required. A crude measure of programmer effort is lines of code; the SystemC implementation of LRU uses 160 lines of code, compared to 130 lines for Sim and 110 lines for straight C++.

5 Discussion

Our experience with Sim has taught us the following five lessons regarding simulation frameworks:

1. Use C++

For a number of reasons, C++ has "The Right Stuff" for developing a simulation framework. First, it is fast. Second, it is object-oriented, and objects are without question the appropriate model for hardware components. Furthermore, well defined construction orders (e.g. base objects before derived objects) allow the framework to automatically deduce the component hierarchy. Third, templated classes allow abstractions such as inputs and outputs to be implemented for arbitrary data types in a clear and intuitive manner. Fourth, macros hide the heavy machinery of the framework behind short, easy to use declarations. Fifth, the preprocessor permits different versions of the same code to be compiled. In particular, debugging mechanisms such as timestamps can be removed in the release build, resulting in an executable whose speed rivals that of straight C++. Sixth, operator overloading allows common constructs to be expressed concisely, and typecast overloading allows the framework's abstractions to be implemented transparently. Finally, C++ is broadly supported and can be compiled on virtually any platform. To our knowledge, no other language has all of these features.

2. Use timestamps

Silent failures are the arch-nemesis of computer programmers. Using timestamped nodes in conjunction with automatically generated hierarchical component names, the Sim framework was able to essentially eliminate all three of the common errors described in Section 2.2 by replacing silent failures with meaningful warning messages.

3. Allow inputs/outputs to be directly connected

Using explicitly declared signals to connect component inputs and outputs is familiar to users of existing hardware description languages. However, directly connecting inputs and outputs does not change the underlying hardware model or make the simulator framework any less powerful. Direct connections reduce the amount of programmer effort required to produce a hardware model, and they lead to an extremely efficient implementation of the input/output abstraction.

4. Don't make excessive use of abstractions

The most useful abstractions provided by a simulation framework are components and the interface between then. Within a component, however, further use of abstractions may not add to the modularity of the design or make the transition to silicon any easier. Thus, when simulation speed is a concern, programmers are well-advised to use straight C++ wherever possible. A good example of this principle is the LFSR benchmark. The Sim implementation could just as easily have implemented the shift register internals using a 128-element byte array, as in the C++ implementation. Using the register abstraction slowed down execution significantly, especially in

the debug build. In general, we found the register abstraction to be most useful for implementing clocked outputs, as in the Adder of Figure 1.

5. Pay attention to the details

While the speed and modeling power of a framework is primarily determined by its high-level design, it is the implementation details that programmers need to work with. How easy is it for the programmer to declare and connect components? Can components be inherited? Can they be templated? The answers to questions such as these will determine which programmers will want to use the framework, and how much effort they must expend in order to do so.

Sim is a cycle-based simulator, but four of the lessons apply equally well to event-driven simulators. The exception is timestamps. In a cycle-based simulation, timestamps guarantee consistency between producers and consumers that are activated on *every* clock cycle. In an event-driven simulation, on the other hand, if a producer forgets to set a certain output then the consumer simply may not be activated. This is further complicated by the fact that update methods are only invoked in response to *changing* inputs. Thus, it may be difficult to distinguish between an output that was not set and one whose value did not change. Since timestamps were found to be enormously beneficial from a debugging standpoint, a promising direction for future research would be to investigate similar mechanisms that could be implemented in an event-driven simulation framework.

References

[Arnout00] Dr. Guido Arnout, "SystemC Standard", Proc. 2000 Asia South Pacific Design Automation Conference, IEEE, 2000, pp. 573-577.

[Cyn01] CynApps, "Cynlib Users Manual", 2001

[Gajski00] D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, S. Zhao, SpecC: Specification Language and Methodology, Kluwer Academic Publishers, Norwell, USA, 2000.

[Kogel01] Tim Kogel, Andreas Wieferink, Heinrich Meyr, Andrea Kroll, "SystemC Based Architecture Exploration of a 3D Graphic Processor", Proc. 2001 Workshop on Signal Processing Systems, IEEE, 2001, pp. 169-176.

[Ku90] D. Ku, D. Micheli, "HardwareC – A Language for Hardware Design (version 2.0)", CSL Technical Report CSL-TR-90-419, Stanford University, April 1990.

[Liao97] Stan Liao, Steve Tjiang, Rajesh Gupta, "An Efficient Implementation of Reactivity for Modeling Hardware in the Scenic Design Environment", Proc. DAC '97, pp. 70-75.

[Ramanathan00] Dinesh Ramanathan, Ray Roth, Rajesh Gupta, "Interfacing Hardware and Software Using C++ Class Libraries", Proc. ICCD 2000, pp. 445-450.

[SC01] "SystemC Version 2.0 User's Guide", available at http://www.systemc.org, 2001