The role of Cascade, a cycle-based simulation infrastructure, in designing the Anton special-purpose supercomputers

J.P. Grossman¹,‡, Brian Towles¹, Joseph A. Bank¹, David E. Shaw¹,²,‡

¹D. E. Shaw Research, New York, NY 10036, USA.
²Center for Computational Biology and Bioinformatics, Columbia University, New York, NY 10032, USA.
†Correspondence: JP.Grossman@DEShawResearch.com and David.Shaw@DEShawResearch.com

ABSTRACT
Cascade is a cycle-based C++ simulation infrastructure used in the design and verification of two successive versions of Anton, a specialized machine designed for high-speed molecular dynamics computation. Cascade was engineered to address the size and speed challenges inherent in simulating massively parallel special-purpose machines. It provides a lightweight programming interface, rich debugging support, tight Verilog integration, fast multithreaded execution, and low memory overhead. Here, we describe the core features of Cascade that proved most valuable for our simulation efforts.

Categories and Subject Descriptors
I.6.7 [Simulation and Modeling]: Simulation Support Systems—Environments; B.8.2 [Performance and Reliability]: Performance Analysis and Design Aids

General Terms
Algorithms, Performance, Design, Verification

Keywords
Cascade, Anton, cycle-based simulation, reflection

1. INTRODUCTION
Software simulation is an invaluable tool in the design and verification of complex hardware architectures: simulations written in a high-level language (typically C or C++) offer performance and flexibility far beyond what can be achieved by Register-Transfer Level (RTL) simulation alone. Simulation infrastructures are generally classified as either event-driven or cycle-based. Event-driven simulators are well suited to coarse-grained hardware models, whereas cycle-based infrastructures are often preferred when greater accuracy is desired, and have been widely used in both industry and academia (e.g., [9,12,17,21,23]).

We relied extensively on simulation for the design and verification of Anton 1 [18] and Anton 2 (in development), two massively parallel special-purpose supercomputers for molecular dynamics computation. To address the challenges presented by simulating machines of this scale, we created Cascade—a cycle-based hardware simulation infrastructure. Much like SystemC [19], Cascade is a C++ library in which hardware modules are modeled as classes (referred to herein as “components” to distinguish them from RTL modules) with explicitly declared input and output ports that can be connected to one another at construction time. Cascade, however, was designed with emphasis on performance, programmability, and scalability—key considerations for simulations comprising millions of components and ports.

This paper describes the core features of Cascade that allowed it to meet these design goals and serve as the basis for the Anton simulators. Some of these features—such as reflection, static scheduling, and Verilog co-simulation—exist in other simulation infrastructures, but have been engineered in novel ways within Cascade to minimize programmer effort. Other features are themselves novel, including run-time port validation, component deactivation, and specialized optimizations for synchronous connections. Together, these features reduce the amount of code required to implement hardware models, provide rich debugging support, and mitigate the performance overheads typically associated with high-fidelity cycle-based simulations.

2. DEVELOPMENT OF CASCADE
One of the earliest decisions we faced in the Anton project was whether to use an existing simulation infrastructure or develop our own. Our conclusion was that the benefits of developing an in-house infrastructure tailored to our requirements justified the time and effort required to do so. The result was Cascade, which has proven to be a valuable tool in our hardware design process. A full description of Cascade is provided elsewhere [3]; here we focus on the features that were most important for our simulation efforts.

2.1 Simulation-based methodology
In designing the Anton machines, we made use of simulation in a variety of ways. Initially, simulation was used for architectural exploration, allowing rapid prototyping and performance evaluation for many different design options. Hardware/software co-design was a critical part of this process as we established the boundaries between programmable embedded cores and special-purpose hardware accelerators. The use of a C++ simulation infrastructure permitted full integration of embedded software within the hardware simulator, often using “magic” interfaces that would later be refined into hardware APIs. Software written with these APIs could then either be directly linked into the simulator and executed natively, or cross-compiled and run on instruction set simulators [8].

As the design solidified, we began the work of RTL implementation, while simultaneously updating simulator components to match the interface and behavior of the corresponding Verilog modules. Bit-accurate interfaces allowed simulator components,
already tested within full-machine simulations running embedded software, to be used as golden models in design-verification testbenches [7]. This provided substantial verification that the RTL modules were functioning correctly and would interact properly with one another and with the embedded software.

### 2.2 Simulation requirements

Throughout this paper we report on four common simulation targets: 64- and 512-node configurations of Anton 1 (“A1-64”, “A1-512”), as well as 64- and 512-node configurations of Anton 2 (“A2-64”, “A2-512”). Table 1 lists the numbers of components and ports in these simulations. The scale of the machines being simulated, together with our hardware design methodology, placed a number of requirements on the simulation infrastructure:

**Language.** Our need to integrate embedded software with the machine simulator led us to focus on C++ infrastructure; special-purpose languages such as Esterel [1] were not a viable option.

**Ease of use.** The scale of the simulator effort demanded an infrastructure with minimal programmer overhead and significant debugging support.

**Verilog co-simulation.** We desired the ability to seamlessly integrate simulator models into Verilog testbenches without the need to write specialized data-marshalling functions.

**Performance.** With simulation times measured in hours, good simulator performance was naturally important.

**Checkpointing.** One specific concern was the time required to debug errors in long simulations. We viewed the ability to checkpoint and restore simulations as essential in order to be able to replay failures (either in a debugger or with additional logging enabled) without having to restart the entire simulation.

**Memory efficiency.** The memory overhead of a simulation infrastructure is often inconsequential given the low cost of DRAM. In our case, however, simulations could contain millions of components and tens of millions of ports, so memory efficiency was an important consideration.

In addition to these requirements, we felt that our needs would best be met by a cycle-based (rather than event-driven) infrastructure. First, we wanted certain simulator models to be cycle accurate where timing was critical for either performance estimation or design verification. Second, cycle-based models with bit-accurate interfaces are more amenable to co-simulation with RTL. Finally, it has been observed that cycle-based simulations may offer better performance than event-driven simulations in practice [19].

### 2.3 Overview of Cascade

Each component in Cascade is a C++ class that inherits from a base Component class, and communication between components is effected via input and output ports. Ports can be directly connected to one another, thus obviating the need for explicitly declared signals or wires. A Cascade simulation consists of three phases. In the “construction” phase, components are constructed and connections are established between input and output ports. In the “initialization” phase, Cascade performs significant pre-processing to set up and optimize the simulation. Finally, the actual hardware simulation takes place in the “simulation” phase, which alternates between rising clock edges and combinational evaluation as simulated time progresses.

The behavior of a component is modeled by one or more “update” functions. An update function reads some subset of the component input ports, updates the internal component state, and sends text to a consumer component one character at a time.

**Checkpointing** is instructive to compare it to Cascade on a representative benchmark to understand why we did not feel it would meet our needs. We modeled a simple 8×8 on-chip mesh network in both Cascade and SystemC, with one client per router capable of sending and receiving messages. We then simulated 512 instances of these “chips” for 10,000 clock cycles with randomized network traffic. The simulation comprised a total of 67,072 components and 1,703,936 ports. We used version 2.2.0 of SystemC, and both implementations were compiled using version 4.5.2 of gcc with -O3. Table 2 summarizes the comparison: the Cascade implementation required 39% fewer lines of code, consumed 14.7 times less memory, and ran 14.7 times faster with multithreading.

### Table 1. Number of distinct component types, component instances and port instances for the four simulation targets described in this paper.

<table>
<thead>
<tr>
<th></th>
<th>A1-64</th>
<th>A1-512</th>
<th>A2-62</th>
<th>A2-512</th>
</tr>
</thead>
<tbody>
<tr>
<td># components</td>
<td>70</td>
<td>70</td>
<td>56</td>
<td>56</td>
</tr>
<tr>
<td># ports</td>
<td>23,559</td>
<td>188,423</td>
<td>276,163</td>
<td>2,209,283</td>
</tr>
<tr>
<td># component types</td>
<td>116,992</td>
<td>935,936</td>
<td>1,976,768</td>
<td>15,814,144</td>
</tr>
</tbody>
</table>

*Figure 1. Simple Cascade example showing a consumer component connected to a producer component.*
Table 2. Comparison between SystemC and Cascade for an on-chip network benchmark with 67,072 components and 1,703,936 ports, simulated for 10,000 clock cycles.

<table>
<thead>
<tr>
<th></th>
<th>SystemC</th>
<th>Cascade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of code</td>
<td>361</td>
<td>220</td>
</tr>
<tr>
<td>Memory (MB)</td>
<td>1249</td>
<td>85</td>
</tr>
<tr>
<td>Seconds (single-threaded)</td>
<td>677</td>
<td>76</td>
</tr>
<tr>
<td>Seconds (multithreaded)</td>
<td>N/A</td>
<td>46</td>
</tr>
</tbody>
</table>

Cascade reduces the number of lines of code by automatically constructing component and port names, eliminating the need for explicit signals, and providing connection operations for matching interfaces (see supplemental section S1 for details). The large difference in memory usage is due to component and port overheads. On a 64-bit machine, the per-component overhead in SystemC is over 4500 bytes; in Cascade it is 36 bytes, comprising a virtual function table pointer (8 bytes), “parent”, “first child” and “next sibling” pointers (8 bytes each), and a 4-byte integer containing additional flags and data. Similarly, the per-port overhead in SystemC for a single-byte port is over 380 bytes, compared to 9 bytes in Cascade (1 byte for a separately allocated value shared by connected ports, and 8 bytes for a pointer to this value). This alone makes SystemC infeasible for our simulations; the 512-node Anton 2 simulation would require over 15 GB of memory for the component and port overhead alone.

3. CORE CASCADE FEATURES

This section describes the core features of Cascade that allowed it to meet our simulation requirements.

3.1 Reflection

Reflection refers to a program’s ability to inspect and operate on its own types at run time (e.g., to iterate over the members of a class or structure). In the context of a simulation infrastructure, reflection allows the infrastructure to automatically garner structural information regarding the programmer-supplied hardware models. While some languages including C#, Python and Java provide native support for reflection, C++ does not. In order to benefit from reflection, a simulation infrastructure written in C++ must therefore supply its own implementation.

Cascade uses a set of macros (including, in Figure 1, the macros DECLARE_COMPONENT, COMPONENT_CTOR, Input and Output) to automatically capture the component hierarchy, the set of ports within each component, and the names of all components and ports. These macros are minimally intrusive, and the programmer does not need to explicitly specify port names separately from the port declarations: the macros define functions that are called automatically to register port names with the reflection infrastructure (see supplemental section S2 for implementation details). This technique is quite general, and it could be used to add lightweight reflection to any C++ library.

The component hierarchy is dynamically represented as a tree using the component “parent”, “first child” and “next sibling” pointers. The remaining information is captured in a true type-based reflection system with no per-instance storage overhead, leading to very lightweight component and port implementations.

3.1.1 Benefits of reflection

Cascade’s reflection system greatly enhances debugging support by allowing error and warning messages to specify the full hierarchical name of a component or port. This often exposes the source of a bug without having to step through the simulation in a debugger. Reflection also allows Cascade to automatically generate Value Change Dump (VCD) files tracing all port values within a subtree of the component hierarchy.

3.1.2 Interface binding

One application of Cascade’s reflection system that was particularly valuable to our design verification effort is interface binding, a mechanism that allows Cascade components to be instantiated within Verilog simulations in the same manner as ordinary Verilog modules. If a Verilog wrapper module and a Cascade component have the same interfaces, Cascade can automatically establish a binding between corresponding ports so that data is transparently marshaled between Verilog and C++. We relied on interface binding in order to reuse simulator components as golden models within Verilog testbenches.

Figure 2 illustrates the use of interface binding for a simple adder component. An empty Verilog module is created with the same interface as the C++ component. The body of the Verilog module consists of a single call to $create_cmodule; this Cascade system task creates the component (as specified by the DECLARE_CMODULE macro) then uses reflection information to iterate over the component ports and bind them to the Verilog ports, verifying that the sizes, directions and names all match.

3.2 Static update schedule

As the simulation components are constructed, Cascade keeps track of combinational dependencies between update functions. This defines a directed graph which is topologically sorted to produce a static update schedule. If combinational cycles are

```cpp
class Adder : public Component
{
    DECLARE_COMPONENT(Adder);
    public:
    Adder (COMPONENT_CTOR) {}

    Clock (clk);
    Input (u16, i_a);
    Input (u16, i_b);
    Output(u17, o_sum);

    void update () { o_sum = i_a + i_b; }   

    DECLARE_CMODULE(adder, new Adder);
};
```

Figure 2. Interface binding. (a) C++ ports are automatically bound to corresponding Verilog wrapper module ports. (b) Definition of the Adder component. (c) Verilog wrapper module, which instantiates the Adder component. On a rising clk edge Cascade copies i_a and i_b to C++, then calls Adder::update(), then copies o_sum back to Verilog.
present, then the topological sort fails with a detailed error message, and the developer must modify the implementation to eliminate the cycle. This can be accomplished either by splitting one or more update functions into separate update functions, or by replacing combinational connections with synchronous connections (which will be described in Section 3.2).

3.2.1 Specifying dependencies

In an event-driven simulation infrastructure such as SystemC, input sensitivities must be specified for each process. Because this information alone is insufficient to determine the combinational dependencies between update functions, Cascade requires the programmer to declare both the ports that are read and the ports that are written by an update function. Cascade’s default assumption, which handles the vast majority of cases, is that a single update function named “update” reads all of a component’s inputs and writes all of its outputs; explicit read/write declarations are required only when this assumption does not hold. Reflection is used to obtain the list of component ports, so in the common case no programmer effort is required to specify dependencies. Of the 56 distinct component types modeled for Anton 2, only 8 required explicit read/write declarations.

Referring again to the code in Figure 1, Cascade will automatically detect the update functions Producer::update() and Consumer::update(); furthermore it will assume that the producer update function writes Producer::o_ch and that the consumer update function reads Consumer::i_ch. The connection between these ports therefore creates a combinational dependency between the update functions, so a static update schedule will be generated in which producer::update() is called before consumer::update().

3.2.2 Component deactivation

The use of a static schedule is known to improve simulator performance and avoid “unnecessary wakeups” [2,13,15,17]. Still, calling every update function on every cycle is a significant source of overhead in cycle-based simulations, particularly for components that are quiescent. Cascade addresses this issue by providing every component with an “active” flag; update functions are only called for active components. The programmer can add code to an update function to determine if the component has become quiescent and, if so, deactivate it.

With this optimization in place, it remains to automatically reactivate components based on the values of certain programmer-specified input ports (typically “valid” bits indicating the arrival of new data). A natural approach is to associate activation with port writes, that is, when a producer component sets its “valid” output port, the act of writing the port causes any consumer components to become activated. This approach, however, would increase the execution overhead for all port writes (due to the need to check for components that must be activated) as well as the per-port storage overhead (in order to specify this set of components).

Instead, Cascade leverages both the static update schedule and the knowledge of the set of ports written by each update function: if a component is conditionally activated based on a certain input port, then Cascade checks the value of the port immediately after invoking the update function that writes it. This allows components to be automatically reactivated without increasing the performance or memory overheads associated with ports. It also avoids testing ports when the producer component is itself inactive, thus maintaining the property that the overall simulation overhead is proportional to the number of active components.

Most of the components in our simulations were instrumented to deactivate themselves when quiescent. We found that this optimization eliminated up to 80% of the update function invocations and improved the overall simulation performance by as much as 28% (Table 3).

3.3 Synchronous connections

Another significant source of overhead in cycle-based simulations is the need to copy register values from their input ports to their output ports on every rising clock edge. This has both a performance impact (the time required to perform the copy) and a storage impact (the need to store two copies of each register value). In a statically scheduled cycle-based simulation, where each update function is guaranteed to be called at most once per cycle, it is generally unnecessary to model registers within components: internal state can simply be stored in member variables and modified appropriately by the update functions. Only registers that exist between components need to be explicitly modeled.

Cascade has native support for synchronous connections between ports: if a port A is connected to a port B using the overloaded less-than-or-equal-to operator (“A <= B”), a register is placed between the ports so that writes to A are only visible from B on the next rising clock edge. Internally, storage is allocated for two values, with port A pointing to one value, port B pointing to the other, and a copy occurring on each rising clock edge.

Givings the simulation infrastructure full ownership of registers enables two optimizations to automatically reduce register overhead. First, Cascade leverages the static update schedule to eliminate many explicit registers altogether. If the first write of a register input port always occurs after the last read of the register output port, then the register is unnecessary and can be replaced by a single uncopied value without affecting simulation behavior. Cascade attempts to generate an update schedule that eliminates as many bytes of register storage as possible. Second, Cascade optimizes register value copies by allocating two large identical blocks of memory, one for register input ports and one for register output ports, then using a single memcpy to copy values on the rising clock edge. In addition to dramatically reducing the performance overhead of register copies, this also improves memory efficiency by eliminating the need for a global data structure used to iterate over the registers. These optimizations consistently eliminated 52% of the register storage, and improved overall simulation performance by up to 17% (Table 4).

| Table 3. Performance improvements due to programmer-driven component deactivation optimizations. |
|---------------------------------|----------|----------|----------|----------|
| % updates eliminated | A1-64 | A1-512 | A2-64 | A2-512 |
| % speedup | 33 | 40 | 75 | 80 |

| Table 4. Performance improvements due to automatic register elimination and fast register copies. |
|---------------------------------|----------|----------|----------|----------|
| % register bytes eliminated | A1-64 | A1-512 | A2-64 | A2-512 |
| % speedup | 17 | 17 | 11 | 13 |
3.4 Multithreaded simulation engine

Cascade uses a simple yet effective multithreading strategy: when multiple clock domains have a simultaneous rising clock edge, the clock domains are partitioned among the available threads so that their register copies and combinational evaluations can be performed in parallel on different cores. The simulation itself is deterministic and functionally independent of the number of threads; multithreading is strictly a performance improvement.

This approach requires some cooperation from the programmer in the form of artificially dividing clock domains into two or more domains with identical clocks. So long as there are no combina-
tion of connections between components in the subdivided do-
mains (Cascade requires connections between clock domains to be synchronous), this transformation is straightforward and has no impact on the simulation. In our simulations, there was a natural partition: each node was placed in a separate clock domain.

Multithreading was not implemented in the initial version of Cascade, and is only supported within our Anton 2 simulator. On a dual-socket quad-core machine, we observed speedups of up to 6.7 fold with 8 threads (Figure 3). These speedups are much larger than the speedup observed for the network benchmark in Table 2, which contains fairly simple update functions and is memory-bandwidth-limited. The Anton 2 simulator is compute-limited and therefore benefits more from multithreading.

3.5 Debugging support

Strong debugging support was an essential requirement for our development process. Cascade implements three debugging mechanisms that are tailored to hardware simulation: run-time port validation, conditional tracing, and simulation checkpointing.

3.5.1 Port Validation

In debug builds, every Cascade port has an associated “valid” flag, immediately preceding the port value in memory, that is set when the port is written, checked when the port is read, and cleared on each rising clock edge. This mechanism automatically detects a variety of common errors:

*Invalid outputs.* A component’s update function(s) may neglect to set one or more outputs, resulting in undefined values.

*Uninitialized outputs.* An output register that is not initialized during reset will have an undefined value on the first cycle.

*Bad evaluation order.* If components with combinational dependencies are not evaluated in the correct order, then the consumer component will read a stale value from the previous cycle.

*Missing connections.* One or more required connections between ports may be omitted at construction time.

*Incorrect handshaking.* Certain ports may be valid only on certain cycles depending on an interface’s protocol. A producer may neglect to set these ports when it is supposed to, or a consumer may attempt to read these ports when it shouldn’t.

When any one of these errors occurs, Cascade produces a warning specifying the full hierarchical name of the invalid port, from which the source of the problem can be quickly deduced. Without port validation, these errors tend to produce silent failures that are difficult to diagnose and correct. A cycle-based infrastructure supports port validation by providing a well-defined point in time (the rising clock edge) at which the port valid flags should be cleared. In an event-driven simulation, by contrast, there is no way to distinguish a port whose value is stale from a port whose value is simply unchanged.

3.5.2 Tracing

One of the most basic debugging techniques is using printf statements to generate output that can be inspected to determine the source of a problem. Cascade extends this technique for hardware simulations with tracing, which is conditional output that can be enabled on a per-component basis. For example, if an error occurs within a specific component (identified by the error message), then the simulation can be re-run with tracing enabled for that component. All trace output generated by a component is prefixed with both the current simulation time and the component’s name.

3.5.3 Simulation checkpointing

Cascade supports simulation checkpointing wherein the entire state of a simulation can be stored to or retrieved from a compressed archive. Furthermore, simulation checkpoints are compatible across debug and release builds. A long simulation can be run in release mode generating periodic checkpoints; one of these checkpoints can then be restored within a debugger in order to inspect simulation state or step through an event of interest.

All port and register state is automatically archived; the programmer is only responsible for archiving internal state by implementing an archive() function for each component. These functions are called automatically by Cascade whenever a simulation is written to or read from a checkpoint.

4. RELATED WORK

SystemC is a widely used simulation infrastructure, but it is known to have performance issues due to its focus on generality [1,4,6,15,22]. In [15] the performance of SystemC was improved using “acyclic scheduling”, which attempts to statically schedule process wakeups based on the graph of combinational dependencies, relying on the event-driven SystemC engine to handle cycles in this graph. A similar technique was proposed in [10] and used in the Liberty Simulation Environment [14]. In [13], combinational cycles were aggressively eliminated using a tool that parses and automatically rewrites SystemC modules, splitting larger processes into multiple smaller ones.
Requiring the programmer to eliminate all combinational cycles from the dependency graph allows the use of a fully static schedule, further improving performance by simplifying the simulation infrastructure and guaranteeing that each process is woken up at most once per clock cycle [2,16,17]. Cascade builds on this approach with support for component deactivation, and by leveraging the static schedule to automatically eliminate registers.

The use of reflection within a simulation infrastructure was proposed in [5] and implemented by providing component descriptions in an interface-description language. SystemC provides a heavyweight instance-based (rather than type-based) naming mechanism that requires the programmer to explicitly supply string names to port constructors. By contrast, Cascade’s reflection system is automatic and minimally invasive, requiring only a slight modification to the port declaration syntax.

5. DISCUSSION

Cascade played a central role in the development of both Anton 1 and Anton 2. For each machine, we used our Cascade-based simulator for architectural exploration and validation, performance estimation, design verification, and embedded software development. Cascade’s support for Verilog co-simulation was particularly valuable. In addition to reusing simulator components as golden models within Verilog testbenches, we were able to run full 512-node machine simulations with one C++ node replaced by an RTL node. Running our molecular dynamics software in this mixed simulation environment provided a more stringent test of the hardware than block-level testbenches, and uncovered a number of hardware and software defects.

Not all of the design requirements listed in Section 2.2 were evident at the outset of the project, and a number of design tradeoffs were made as Cascade underwent several major revisions. For example, earlier versions of Cascade supported threaded components (similar to SystemC’s SC_THREAD) based on the QuickThreads library [11]. The use of QuickThreads, however, is incompatible with checkpointing. As our need to debug long simulations grew, we abandoned threaded components in favor of the ability to save and restore simulations.

The Cascade features discussed in Section 3 are those that were found to be most useful for our design efforts; all of them could be incorporated into various other simulation infrastructures. Component deactivation, synchronous connections and port validation are more specific to cycle-based simulation, while component deactivation, synchronous connections and port validation are more specific to cycle-based simulation, while component deactivation, synchronous connections and port validation are more specific to cycle-based simulation.

The Cascade features discussed in Section 3 are those that were found to be most useful for our design efforts; all of them could be incorporated into various other simulation infrastructures. Component deactivation, synchronous connections and port validation are more specific to cycle-based simulation, while component deactivation, synchronous connections and port validation are more specific to cycle-based simulation. Component deactivation, synchronous connections and port validation are more specific to cycle-based simulation.

6. REFERENCES

S1. Comparison with SystemC
We modeled an on-chip mesh network benchmark in both Cascade and SystemC, and found that the Cascade implementation required fewer lines of code (220) than the SystemC implementation (361). Here we compare portions of the source code for the two implementations to explain this difference.

1. Packet header port type. In Cascade, all port types must be “bags of bits”: they cannot have constructors, destructors, or virtual functions. One of the ways in which SystemC is more general is that any C++ type may be used as a port type. To support this generality, all port types must define a comparison operator, a streaming operator, and a sc_trace function. Our benchmark includes a PacketHeader port type, defined in the Cascade implementation as

```cpp
struct PacketHeader {
    byte dest_x;
    byte dest_y;
};
```

The corresponding SystemC definition, including the required operators, is as follows:

```cpp
struct PacketHeader {
    byte dest_x;
    byte dest_y;

    bool operator== (const PacketHeader &rhs) const
    {
        return dest_x == rhs.dest_x &&
        dest_y == rhs.dest_y;
    }
};
```

These features include sc_trace and the streaming operator are unused, but the (null) implementations are required for the code to compile.

2. Tracing and assertions. Cascade is built on a library that adds context names to assertion failure messages and supports conditional logging (tracing) based on context names. We incorporated these features in the SystemC implementation by defining an sc_component base class that inherits from both sc_module and the library’s TraceContext helper class:

```cpp
class sc_component : public sc_module, public TraceContext {
    public:
        sc_component (sc_module_name name) :
            sc_module(name),
            TraceContext((const char *) name)
        {
        }
    
    int credits;
    
    Input(bit, i_valid);
    Input(PacketHeader, i_header);
    Input(uint64, i_data);

    Output(bit, o_valid);
    Output(PacketHeader, o_header);
    Output(uint64, o_data);

    sc_trace and the streaming operator are unused, but the (null) implementations are required for the code to compile.

3. Network port interface. In the Cascade implementation, the credit-based network port interface is defined as follows:

```cpp
struct NetPort : public Interface {
    DECLARE_INTERFACE_WITH_CTOR(NetPort);

    NetPort (INTERFACE_CTOR) : credits(4) {
        o_valid.setType(PORT_PULSE);
        o_credit.setType(PORT_PULSE);
    }

    void disable (){
        i_valid.wireToConst(0);
        i_credit.wireToConst(0);
    }

    Input(bit, i_valid);
    Input(PacketHeader, i_header);
    Input(uint64, i_data);

    Output(bit, o_valid);
    Output(PacketHeader, o_header);
    Output(uint64, o_data);

    int credits;

    syncConnect(m_routers(x-1,y).port[DIR_XPOS],
                m_routers(x,y).port[DIR_XNEG]);
```

In the constructor, the o_valid and o_credit outputs are given the type PORT_PULSE, which causes them to be reset to zero on every rising clock edge. The disable() function is used to tie off unconnected network ports by forcing the i_valid and i_credit inputs to zero.

The Interface base class, along with the macros DECLARE_INTERFACE_WITH_CTOR and INTERFACE_CTOR, define a Cascade interface. Cascade provides connection operations for matching interfaces, so for example the synchronous on-chip network connections in the x dimension are established using the following code:

```cpp
syncConnect(m_routers(x-1,y).port[DIR_XPOS],
m_routers(x,y).port[DIR_XNEG]);
```

The System C network port interface definition, including helper functions to connect and disable network ports, is as follows:

```cpp
$ sc_router
Error: Client(0,3)
Assertion failed: m_inSeq[idx] == int(i_data)
sc_router.cpp:92: void Client::update() 

We re-ran the simulation with tracing enabled for the client identified in the error message:

```cpp
$ sc_router -trace "Client(0,3)"
Client(0,3): [7 ns] Sending message to 23 (7,2)
Client(0,3): [9 ns] Received message from 16 for 24
Client(0,3): [10 ns] Sending message to 58 (2,7)
Client(0,3): [13 ns] Received message from 2 for 24
Client(0,3): [14 ns] Received message from 39 for 24
Client(0,3): [15 ns] Received message from 2 for 0
Error: Client(0,3)
Assertion failed: m_inSeq[idx] == int(i_data)
sc_router.cpp:92: void Client::update() 

This revealed that the client, whose numerical ID is 24, was receiving a message that should have been delivered to Client(0,0). From this starting point we were able to track down the source of the error—a bug in the comparison operator that was causing a stale packet header to be reused in the network.
There are three sources of overhead in this code, all of which Cascade eliminates. First, the port names must be explicitly specified in the NetPort constructor. Second, signals are required to connect network ports or to tie off disabled network ports. Third, helper functions must be provided to bind signals to ports and to connect two ports.

4. **Router model.** The two router class definitions are very similar, as can be seen in a side-by-side comparison:

### SystemC
```cpp
class Router : public sc_module
{
public:
    // DECLARE_COMPONENT(Router);
public:
    Input(byte, i_x);
    Input(byte, i_y);
    RouterPort port[5];
    Router (COMPONENT_CTOR);
    void update ();
private:
    int m_rrPriority;
};
```

### Cascade
```cpp
class Router : public Component
{
DECLARE_COMPONENT(Router);
public:
    Input(byte, i_x);
    Input(byte, i_y);
    RouterPort port[5];
    Router (COMPONENT_CTOR);
    void update ();
private:
    int m_rrPriority;
};
```

There are two differences worth noting. First, the Cascade model contains a static array of router ports (the RouterPort class inherits from NetPort), whereas the SystemC model contains an array of pointers: each router port must be individually constructed so that the interface names can be provided. This is also the reason that the SystemC model requires a destructor (to delete the router ports). Second, an explicit clock port is unnecessary within Cascade components; a component with no clock port is automatically placed in the same clock domain as its parent.

The `update()` functions in the two implementations are nearly identical; the only difference is that the SystemC implementation explicitly clears the `o_valid` and `o_credit` outputs on every cycle. The destructor in the SystemC implementation simply deletes the router ports. The constructors are as follows:

```
// Cascade
Router::Router (IMPL_CTOR) : m_rrPriority(0)
{
}
// SystemC
Router::Router (sc_module_name name) :
Component(name),
i_x("i_x"),
i_y("i_y"),
clk("clk"),
m_rrPriority(0)
{
for (int i = 0 ; i < 5 ; i++)
port[i] = new RouterPort(str("Port%d", i));
```

Most of the extra code in the SystemC implementation is related to port and interface names. In addition, the `update` function must be explicitly registered and made sensitive to the rising clock edge; in Cascade this is automatic.

**Summary.** Table S1 gives a detailed breakdown for the extra 141 lines of code in the SystemC implementation.

<table>
<thead>
<tr>
<th>Description</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component/port names</td>
<td>56</td>
</tr>
<tr>
<td>Signals</td>
<td>27</td>
</tr>
<tr>
<td>Interface connections</td>
<td>16</td>
</tr>
<tr>
<td>User-defined port type</td>
<td>14</td>
</tr>
<tr>
<td>Tracing/assertions</td>
<td>11</td>
</tr>
<tr>
<td>Method declarations</td>
<td>8</td>
</tr>
<tr>
<td>Other</td>
<td>9</td>
</tr>
</tbody>
</table>

Table S1. Breakdown of additional lines of code required in the SystemC implementation of the network benchmark.
S2. Lightweight reflection for C++ classes

Reflection mechanisms for C++ typically require the programmer to explicitly specify class-member names separately from the class-member declarations. In Cascade, by contrast, the port macros simultaneously declare the ports and register them with the reflection infrastructure. This is accomplished by defining functions that are automatically called before a component is constructed. Defining these functions is easy, whereas calling them automatically requires some engineering.

Consider a simple adder with two inputs and one output:

```cpp
class Adder : public Component
{
    static void preConstruct ()
    {
        _preConstruct0();
    }
    static void _preConstruct (_Index<_Counter<__LINE__,0>::count> *descriptor)
    {
        static void _preConstruct2 ()
        {
            _preConstruct1();
        }
        static void _preConstruct1 ()
        {
            _preConstruct0();
        }
        static void _preConstruct0 ()
        {
            register reflection information for i_a
            _preConstruct1();
        }
        Input<int> i_a;
        Input<int> i_b;
        Output<int, o_sum> o_sum;
    }
};
```

Our strategy is to have the preprocessor transform the above component definition into something resembling the following:

```cpp
class Adder : public Component
{
    static void preConstruct ()
    {
        _preConstruct0();
    }
    static void _preConstruct3 (_) {}
    Input<int> i_a;
    Input<int, i_b>
    Output<int, o_sum>;
}
```

The key elements are (1) a sequence of _preConstruct functions, one per port, such that each one calls the next; (2) an empty _preConstruct function called by the last port’s function to terminate the chain of function calls; and (3) a top-level preConstruct function invoked by the infrastructure that calls _preConstruct0() to register all ports.

The actual macro definitions used to convert this strategy into a working implementation are shown in Figure S1. DECLARE_REFLECTION_FUNCTIONS is a helper macro that appears in the expansion of DECLARE_COMPONENT, and DECLARE_NAMED_PORT is a helper macro that appears in the expansions of the Input and Output port macros.

The templated _Counter structure defines a compile-time function that maps a port’s line number to its zero-based index within the component. In particular, the index of a port is given by _Counter<_LINE_, 0>::count.

The templated _Index<n> structure is used to define a different function for each port. Instead of defining functions with different names, the macros overload the same function name (_preConstruct) with different argument types (pointers to _Index<n>). Each function calls the next simply by casting the descriptor argument to a pointer to _Index<n+1>.

The last call to _preConstruct() does not match the signature of any port function and is therefore captured by the empty version defined by DECLARE_REFLECTION_FUNCTIONS.

Within each _preConstruct function, the getsize struct is used to support statically-sized single-dimensional arrays, e.g.,

```cpp
Input(int, i_data[4]);
```

In particular, the expression offsetof(getsize, name) statically evaluates to the size of the array, or to zero if the port is not an array.

The descriptor object is responsible for assembling reflection information for the component type. Each port registers itself with this object within its _preConstruct function, supplying the byte offset of the port within the component, the name of the port, the port array size (or zero for a scalar port), and the size of the port in bytes.

These macros place minimal burden on the programmer and could easily be adapted to any C++ library that would benefit from reflection.

```cpp
#define DECLARE_REFLECTION_FUNCTIONS()
    template <int line, int x> struct _Counter { enum { count = _Counter<line-1,x>::count }; };
    template <int x> struct _Counter<__LINE__,x> { enum { count = -1 }; };
    template <int n> struct _Index : public Cascade::InterfaceDescriptor {}; static void _preConstruct (...) {}
    template <int line, int> struct _Index<_Counter<line-1,line>>, Cascade::InterfaceDescriptor *descriptor) {
        _preConstruct(_Index<0>*) descriptor);
    }
```

```cpp
#define DECLARE_NAMED_PORT(type, name)
    type name;
    template <int x> struct _Counter<__LINE__,x> { enum { count = _Counter<__LINE__-_1,x>::count + 1 }; };
    static void _preConstruct (Cascade::InterfaceDescriptor *descriptor) {
        struct getsize { byte name; }
        descriptor->addPortName(offsetof(_thistype, name), #name, offsetof(getsize, name), sizeof(type));
        _preConstruct(_Index<_Counter<_LINE_,0>::count + 1>* descriptor);
    }
```

Figure S1. Reflection macros. DECLARE_REFLECTION_FUNCTIONS defines the helper structures _Counter and _Index, as well as the main _preConstruct function. DECLARE_NAMED_PORT defines the per-port _preConstruct function that registers the port with the reflection infrastructure, then calls the next port’s function.