Hierarchical Simulation-Based Verification of Anton, a Special-Purpose Parallel Machine

J.P. Grossman, John K. Salmon, C. Richard Ho, Douglas J. Ierardi, Brian Towles,
Brannon Batson, Jochen Spengler, Stanley C. Wang, Rolf Mueller, Michael Theobald,
Cliff Young, Joseph Gagliardo, Martin M. Deneroff, Ron O. Dror and David E. Shaw*
D. E. Shaw Research, New York, NY 10036, USA
*{JP.Grossman, John.Salmon, Richard.Ho, Doug.Ierardi, Brian.Towles,
Brannon.Batson, Jochen.Spengler, Stan.Wang, Rolf.Mueller, Michael.Theobald,
Cliff.Young, Joe.Gagliardo, Marty.Deneroff, Ron.Dror, David.Shaw}@DEShawResearch.com**

***Abstract*—One of the major design verification challenges in the development of Anton, a massively parallel special-purpose machine for molecular dynamics, was to provide evidence that computations spanning more than a quadrillion clock cycles will produce valid scientific results. Our verification methodology addressed this problem by using a hierarchy of RTL, architectural, and numerical simulations. Block- and chip-level RTL models were verified by means of extensive co-simulation with a detailed C++ architectural simulator, ensuring that the RTL models could perform the same molecular dynamics computations as the architectural simulator. The output of the architectural simulator was compared to a parallelized numerical simulator that produces bitwise identical results to Anton, and is fast enough to verify the long-term numerical stability of computations on Anton. These explicit couplings between adjacent levels of the simulation hierarchy created a continuous verification chain from molecular dynamics to individual logic gates.**

I. Introduction

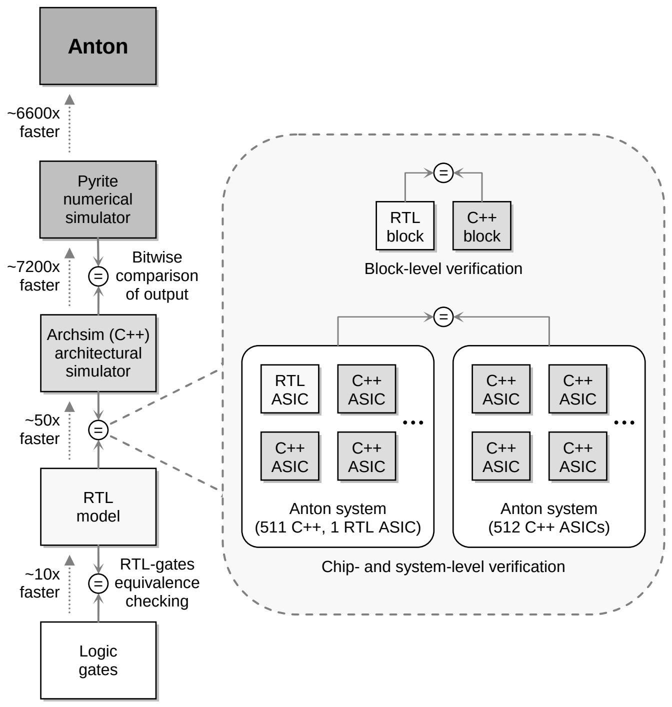
Anton is a special-purpose parallel machine, currently under construction, that is intended for use as a computational tool for research on the structural dynamics of biomolecular systems. Anton was designed to dramatically accelerate molecular dynamics (MD) calculations relative to other parallel solutions [1], enabling the atomic-level modeling of proteins and other biological macromolecules over timeframes far beyond the current state of the art. Such computations should in principle allow qualitative advances in our scientific understanding of biological systems, and may ultimately prove useful in the process of drug discovery and development.

A full Anton machine comprises 512 application-specific integrated circuits (ASICs) connected in a three-dimensional torus network. In order to achieve its high performance, the Anton ASIC implements algorithms and numerical formats that differ from those that have been extensively validated within standard parallel MD codes. The ultimate goal of Anton's verification effort was thus to show not only that the register-transfer level (RTL) model correctly implements Anton's design specification, but also that it correctly implements the underlying physics of an MD calculation. The substantial resource investment required to produce specialized hardware demanded pre-silicon verification that Anton can produce valid scientific results.

This design verification goal presented a formidable challenge. As with other recent system-on-a-chip ventures [2], [3], extensive block-, chip- and system-level verification were essential. These tasks alone required considerable effort due to the complexity of the Anton ASIC, which is a heterogeneous, high-performance multiprocessor with a large number of specialized functional units. Even system-level verification, however, was insufficient to demonstrate the validity of MD computations on Anton. In particular, the detection of undesirable numerical artifacts within an implementation of MD requires much longer computations than can be modeled within an architectural simulation of a 512-node Anton machine. We therefore relied on a hierarchy of RTL, architectural and numerical simulations to form a continuous verification chain linking Anton's gate-level implementation to the numerical stability of long MD computations (Fig. 1). The overall design verification process also included specialized methods such as model checking and asynchronous clock-domain crossing analysis, but in this paper we focus on our simulation-based methodology.

At the bottom of the hierarchy, certain gate-level simulations were run to verify reset behaviour, but we primarily relied on logic equivalence checking to formally verify synthesized logic gates against Anton's RTL model. Next, the RTL model was verified at both the block and chip level using a detailed C++ architectural simulator for Anton, which we refer to as *Archsim*. At the block level, Archsim components were used as golden reference models within RTL testbenches driven by constrained random stimuli. At the chip level, mixed-level simulations were run with 511 C++ ASIC models and one full-chip RTL ASIC model. Combined with RTL assertions and coverage-driven completeness criteria, these test environments demonstrated that the ASIC is capable of correctly running the MD software and producing results identical to those of Archsim. Finally, a parallelized numerical simulator, named *Pyrite*, was de-

* Correspondence to David.Shaw@DEShawResearch.com. David E. Shaw is also with the Center for Computational Biology and Bioinformatics, Columbia University, New York, NY 10032.



Anton
~6600x faster
Pyrite numerical simulator
~7200x faster
Bitwise comparison of output
Archsim (C++) architectural simulator
~50x faster
RTL model
~10x faster
RTL-gates equivalence checking
Logic gates
RTL block
C++ block
Block-level verification
RTL ASIC
C++ ASIC
Anton system (511 C++, 1 RTL ASIC)
Anton system (512 C++ ASICs)
Chip- and system-level verification

Fig. 1. Verification chain from long MD computations to logic gates.

veloped that exactly reproduces the computations and numerical formats implemented by Anton, but runs over three orders of magnitude faster than Archsim. Pyrite can perform MD computations that are long enough to test numerical stability, thus providing the top level of design verification for Anton.

In addition to discussing our overall simulation-based verification methodology, this paper also describes our technique for C++/Verilog co-simulation. We developed a cycle-based hardware simulation infrastructure that supports efficient and flexible interchange of arbitrary C++ and Verilog models at both the block and chip level. This infrastructure effectively removes the boundary between C++ and Verilog, greatly facilitating the creation of test environments containing a mixture of Archsim and RTL models.

A. Related Work

The multiple levels of simulation shown in Fig. 1 are similar to those use to verify the IBM eServer z900 [4], but with an additional numerical level, and with explicit comparisons between all adjacent levels to form a continuous verification chain throughout the simulation hierarchy. Comparisons between Archsim and RTL models in particular relied heavily on the support for C++/Verilog co-simulation within our simulation infrastructure.

Co-simulation has been used in a number of previous chip design endeavors both to verify the interactions between software and hardware, and to perform design verification with mixed-level simulations. When different languages are used for RTL and behavioral models, multiple simulations are often run in different processes using various forms of inter-process communication (IPC) such as remote procedure calls, Berkeley sockets, and Microsoft's Component Object Model; these approaches are described in [5], [6], and [7] respectively. Hoffman et al. used C language interfaces (such as VHDL's Foreign Language Interface) in conjunction with a bus protocol layer to perform mixed-level simulation within a single process [8]. This methodology was applied to both block- and chip-level verification for an ATM switch, and has much in common with our approach to verification at these levels.

A single-language co-simulation methodology has been gaining in popularity wherein the C++ SystemC library [9] is used for both behavioral and RTL models. In [10], the SystemC bus functional model API is used for co-simulation of a processor instruction set simulator (ISS) with a hardware model. In [11], ISS co-simulation is performed by either directly embedding the ISS in a SystemC simulation, or by using the gdb protocol to communicate with the ISS within a SystemC wrapper. More general couplings between behavioral and RTL models are explored in [12] by implementing a mixed-level adapter channel that translates specific data types to and from bit-level representations.

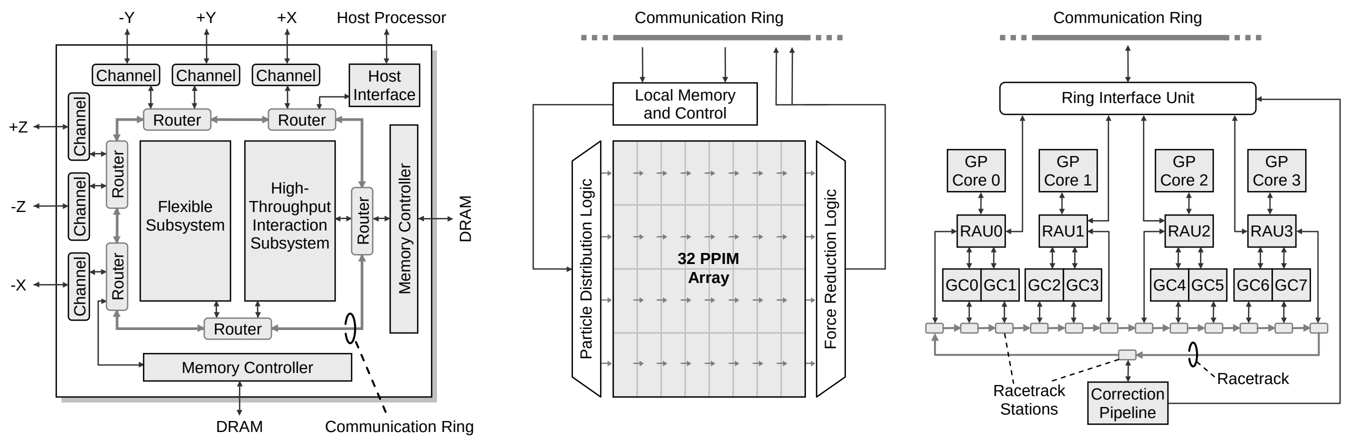
We extend this previous work on co-simulation with an infrastructure that allows C++ and Verilog models to be connected within a single process, and that efficiently marshals data between the C++ and Verilog simulation domains on a cycle-by-cycle basis.

II. The Anton ASIC

In this section we present a brief overview of the Anton architecture. A more detailed explanation of the architecture and how it is used to perform MD computations can be found in [1], while the individual computational subsystems are described in [13] and [14].

Anton is designed to accelerate MD computations, which model the motion of a collection of atoms according to Newton's laws of physics. An MD computation divides continuous time into a sequence of discrete *time steps*, each of which consists of two phases. In the *force calculation* phase, the total force on every atom is computed; this force depends only on the positions of the atoms and is composed largely of pairwise electrostatic forces. In the *integration* phase, the force on every atom is used to update atom positions and velocities. Typically, each time step represents a few femtoseconds of physical time; Anton is intended to run MD computations for up to milliseconds of physical time (close to a trillion discrete time steps).

Anton achieves the speed required for computations of this scale through a combination of specialized hardware, high-bandwidth communication, and fine-grained parallelism. The Anton ASIC consists of two main computational subsystems: the *high-throughput interaction subsystem* (HTIS) [13], which computes pairwise interactions, and the *flexible subsystem* [14], which contains a number of programmable processors (Fig. 2). Two memory controllers are



(a) (b) (c)

Fig. 2. (a) Anton ASIC. (b) High-throughput interaction subsystem. (c) Flexible subsystem.

connected to off-chip dynamic random-access memory (DRAM). A host interface communicates with an external host processor used to control and monitor the ASIC, and six communication channels connect the ASIC to its neighbors in the three-dimensional torus network. These components communicate with one another by sending packets over a bidirectional on-chip communication ring, which consists of six identical routers connected in a loop. Two packet types are of particular relevance to the MD computation and hence to verification: a *position packet* is used to send up to 16 atom positions to the computational units that compute forces, and a *force packet* is used to return the computed forces.

The bulk of the MD calculations are performed by the HTIS, which contains an array of 32 specialized *pairwise point interaction modules* (PPIMs). A systolic network distributes atom position data to the inputs of the 32 PPIMs and aggregates result data from their outputs. Despite being largely composed of special-purpose hardware, the HTIS is highly configurable: it supports 16 different basic modes of operation, the PPIM interaction functions are table-driven, and there are over one hundred hardware configuration registers. This configurability, while desirable for supporting a wide range of MD computations, results in a combinatorial explosion of possible HTIS settings, which makes design verification particularly challenging.

Most of the remaining tasks are performed within the flexible subsystem. Eight internally developed *geometry cores* (GCs) are used for numerical calculations; each GC is a 128-bit, dual-issue, 4-way SIMD processor. Bookkeeping tasks and overall coordination are performed by four general-purpose (GP) cores, each of which is a customized Tensilica LX processor [15]. Each GP core is paired with a *remote access unit* (RAU) that can autonomously send and receive data. The flexible subsystem also contains a *correction pipeline* (CP), a hardware pipeline that computes prescribed adjustments to the force between certain pairs of atoms. Finally, a high-bandwidth *racetrack* serves as a local interconnect between the four GP cores, the eight GCs, and the CP, each of which is connected to a *racetrack station*.

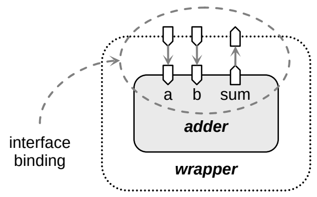
III. Architectural Simulation and RTL Verification

Anton's C++ architectural simulator, Archsim, was instrumental throughout the hardware-software codesign process that was used to develop Anton. As the MD computation was partitioned between specialized hardware and embedded software (i.e. the software that runs on the embedded processors), detailed simulator models were constructed for each of the hardware components described previously. This was essential for verifying Anton at the architectural level, and ensuring that the individual hardware blocks could cooperate to correctly and efficiently compute an MD time step.

Our RTL design verification strategy involved reusing, to the greatest extent possible, the considerable investment in Anton's detailed architectural simulator. This reuse was achieved in two ways. First, individual simulator components were instantiated as reference models within block-level RTL testbenches. Second, a full RTL ASIC model was instantiated as a replacement for a C++ ASIC model within Archsim, and the resulting mixed C++/RTL simulation was used to run MD computations. This co-simulation of C++ and RTL models established a strong verification link between the corresponding levels of our simulation hierarchy.

A. Interface Binding

The effective use of co-simulation techniques requires an infrastructure that supports efficient communication and synchronization between C++ and RTL models. To satisfy this requirement, Archsim was built using an internally developed cycle-driven hardware simulation infrastructure that allows a near-seamless interaction between C++ and Verilog within a single process. The infrastructure contains a class



```
// C++ interface
struct IAdder : public Interface
{
  DECLARE_INTERFACE(IAdder);
  Input<u16>  in_a;
  Input<u16>  in_b;
  Output<u16> out_sum;
};
```

(a) Interface binding; sample interface.

```
// C++ component
class Adder : public Component, public IAdder {
  DECLARE_COMPONENT(Adder);
public:
  Adder (COMPONENT_CTOR) {}
  void update () { out_sum = in_a + in_b; }
};

// Verilog wrapper
module adder_shell (in_a, in_b, out_sum);
  input  [15:0] in_a;
  input  [15:0] in_b;
  output [15:0] out_sum;
  initial $create_component(in_a, "adder");
endmodule

// $create_component "adder" implementation
Adder *adder = new Adder;
BindInterface(adder, module_handle);
```

(b) Verilog wrapper around C++ adder.

```
// Verilog module
module adder (
  in_a, in_b, out_sum
);
  input  [15:0] in_a;
  input  [15:0] in_b;
  output [15:0] out_sum;
  assign out_sum = in_a + in_b;
endmodule

// Module registration
adder a1(wire_a, wire_b, wire_sum);
initial $register_component(a1, "adder");

// C++ wrapper instantiation
IAdder *adder =
 VerilogComponent<IAdder>::Create("adder");
```

(c) C++ wrapper around Verilog adder.

Fig. 3. (a) C++ interface for a simple adder. (b) C++ implementation within a Verilog wrapper. Update() is called automatically by the simulation infrastructure on a rising clock edge. The Verilog wrapper module calls $create_component to instantiate the C++ implementation, passing the name of the component to instantiate ("adder") and a port used to obtain the wrapper module handle. The $create_component "adder" code creates a C++ adder and binds the interfaces. (c) Verilog implementation within C++ wrapper. The module instance is registered as "adder"; the C++ wrapper can then be created by supplying the interface (IAdder) and the module name ("adder").

library that includes hardware interfaces consisting of typed ports, and it supports automatic *interface binding* between C++ models and Verilog modules: if a C++ model has an interface that exactly matches a portion of a Verilog module's interface with the same ports in the same order (with the exception of clock and reset which are handled separately), an initialization-time function can establish a binding between corresponding ports.

At simulation time, the infrastructure transparently marshals data between the Verilog and C++ ports using the standard programming-language interface (PLI) of Verilog simulators. On each clock cycle, the values of the bound Verilog ports are obtained in binary and copied directly to the corresponding C++ ports. Next, update functions for the C++ models are called to simulate a single clock cycle. Finally, the values of the bound C++ ports are retrieved and written in binary to the corresponding Verilog ports.

Interface binding is an efficient mechanism for coupling C++ and RTL simulations as it avoids the overheads associated with IPC or an explicit protocol layer. It also minimizes the programmer effort required to create and maintain an association between C++ and Verilog models; the only requirement is that the interfaces be kept consistent.[1]

The primary use of interface binding is to create a mixed-level simulation by either instantiating a C++ component within an RTL simulation, or instantiating an RTL component within a C++ simulation. In the former case, the component is implemented in C++, and a Verilog wrapper module is defined with the same interface (Fig. 3a,b). The wrapper module can then be instantiated within the Verilog simulation in exactly the same manner as a pure Verilog module. Similarly, when a C++ wrapper is created around a Verilog implementation (Fig. 3c), the wrapper can be used in the same manner as a pure C++ component.

[1] Individual port binding is also supported to handle those cases where it is difficult to maintain consistency between the C++ and Verilog interfaces.

Interface binding also provides a convenient mechanism for attaching C++ "probes" to an RTL model within a mixed-level simulation. To monitor a set of module interface signals, a C++ component with the appropriate input-only interface (the probe) is instantiated. Then the Verilog module to be monitored is registered as shown in Fig. 3c. Finally, a handle to the Verilog module is obtained, and the interface signals of interest are bound to the interface of the C++ probe. Thereafter, the C++ probe has cycle-by-cycle visibility of the interface signals, and can perform any desired action (e.g., write to a log, test assertions, collect statistics). The probe is non-destructive and does not affect the RTL simulation; one can also create an active C++ component that both reads and writes signals, overriding any values driven from within the RTL simulation.

B. Block-Level Testbenches

The majority of Anton's block-level design-verification (DV) testbenches made use of C++ components from Archsim as reference models. These models were instantiated within Verilog wrappers using interface binding, as described in the previous section and illustrated in Fig. 4. The device under test (DUT) and the reference model were driven by a constrained random stimulus generator within a Vera testbench; the outputs were then compared by the testbench. The memory controller, GC, CP, RAU, HTIS, router and racetrack station were all tested in this manner.

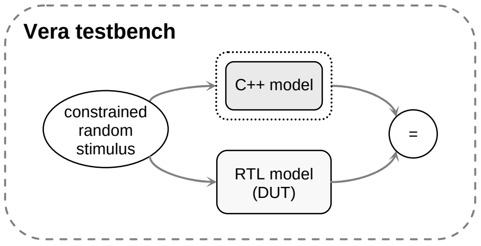


Fig. 4. Block-level testbench with C++ reference model.

This methodology has two main advantages. First, reusing an existing model obviates the need to create yet another model specifically for checking RTL responses within the testbench. Moreover, the C++ reference model of each block has already been debugged as a component of Archsim, largely eliminating the difficult task of simultaneously debugging both the reference model and the DUT (but not completely, as there are certain hardware features that Archsim does not exercise). Without a pre-verified reference model, it is tedious to determine if testbench mismatches are due to a bug in the model or a bug in the DUT. Second, the DUT is compared to a model that is known to work for the target application (MD). This addresses discontinuities between the components of Archsim and their hardware specifications, which can arise when the specification is incomplete or ambiguous, or when problems with a specification are directly addressed within the Archsim component and are not fixed in the specification itself.

Each of the block-level DV testbenches implemented synchronization and result comparison differently, depending on the accuracy of the corresponding C++ reference models. The racetrack station testbench was the simplest as the C++ racetrack model was both bit- and cycle-accurate, allowing for direct output comparisons. The remaining models required a layer of translation to deal with packets, which are transmitted as pointers within Archsim rather than as individual flits. This translation only required a few lines of C++ code, and provided Archsim models with the bit-level interface required for the DV testbenches. Most of the models were also not cycle-accurate, so validation within the testbenches was performed at the transaction level (e.g., a packet send). For the RAU testbench in particular, it was necessary to account for possible reordering of output events between the RTL and Archsim models.

C. *Full-ASIC Testing*

The next level in Anton's simulation-based design verification hierarchy involved running actual MD embedded software on an entire RTL ASIC model. These tests exercised top-level connectivity logic not covered by block-level tests, and additionally verified interactions among hardware components that are difficult to adequately model in block-level tests. Running the embedded software also exposed some gaps in the block-level tests. The configuration space for most of the hardware blocks is enormous, and there are complex inter-dependencies between input data whose arrival could be spread out over thousands of clock cycles. This made it impossible for testbenches driven by constrained random stimuli to fully explore the state space relevant to the target MD application. The full-ASIC tests with the embedded software therefore supplemented block-level design verification by testing additional states that arise during the MD computation.

A full Anton machine contains 512 ASICs, and simultaneously simulating each of these ASICs as an RTL model would be far too expensive. Instead, our approach was to run a mixed-level simulation by replacing a single C++ ASIC model in Archsim with an RTL ASIC model; the RTL model was placed within a C++ wrapper using interface binding. We found the RTL ASIC model to be roughly 50 times slower than the C++ ASIC model, so the speed of a mixed-level simulation (measured in cycles per second) is in fact limited by the 511 C++ models, not the single RTL model. The total number of simulated cycles, on the other hand, is larger due to explicit modeling within the RTL ASIC of embedded processor initialization and cache misses. Using mixed-level simulations, we were able to run up to 100 MD time steps with a full-ASIC RTL model, which took two and a half days and required simulating over three million clock cycles.

A mixed-level simulation is validated by first running the same MD computation in Archsim with only C++ ASIC models. In this reference run, a packet log is generated for all position and force packets sent from or received by subsystems on the ASIC of interest. The MD computation is then rerun in the mixed-level simulation, and the corresponding packet log is generated by using interface binding to attach a C++ packet logger class to the routers of the RTL ASIC model. Once both simulations have completed, the two packet logs are compared, and any differences are flagged as errors. The packet log comparison is non-trivial because the two simulations have very different timings, resulting in different packet orders and often even different groupings of position and force data within the packets. A validation script was used to parse the packet contents from both runs so that comparisons could be made on a per-atom basis rather than a per-packet basis. This approach is more difficult than simply comparing the final outputs of both computations, but it is much more valuable from a debugging standpoint because miscomparisons isolate RTL model problems both in *origin* (which subsystem was the source of the problem) and in *time* (at what point during the MD computation the error occurred).

As with the block-level tests, we again generated constrained random inputs by randomizing both the MD computation and the machine configuration. In this case the ASIC and the embedded software were, together, treated as the DUT, and the randomization was over molecular systems, MD operating parameters, machine size (number of nodes), and the location of the RTL ASIC model within the simulated machine. In addition, artificial network stressors were introduced by delaying packets either entering or leaving the RTL ASIC model according to a randomly-generated delay schedule.

The full-chip test environment was also useful for exercising various ASIC control and debug features in conjunction with the system software that uses them. For example, the JTAG-based gdb stub provided with the Tensilica processors was wrapped in a C++ JTAG interface and then attached to the ASIC's JTAG ports using interface binding.

This allowed xt-gdb (the Tensilica version of gdb) to form remote connections to the Tensilica processors in the RTL ASIC model.

D. *Accelerating ASIC Initialization*

Booting the ASIC and preparing it to run an MD computation takes a significant number of cycles for several reasons. First, nearly two thousand configuration registers must be written to initialize the ASIC, and the hardware mechanism for setting these registers is quite slow. Second, several megabytes of data must be copied into the ASIC's DRAMs in preparation for an MD computation. Third, when the four Tensilica processors in the flexible subsystem boot, they execute reset code that resides off-chip in an uncached region of address space and is fetched over a slow, shared 4-bit bus. Explicitly performing all of these operations in simulation is expensive: all together, initializing a 512-node machine with a single RTL ASIC model takes around 3 days of simulation on a 3 GHz Intel Xeon processor. While it was necessary to occasionally test the full initialization sequence (this was part of a set of regression tests run weekly), we made use of a number of "back-door" techniques to accelerate initialization for the majority of our full-chip simulations.

To speed up writes to the configuration registers (writes which are normally serialized), interface binding was used to connect multiple instances of a simple C++ driver class to the RTL modules containing these configuration registers. This allowed the modules to be initialized in parallel and at high bandwidth, reducing the configuration register initialization time by roughly 30-fold. The explicit copying of data into the ASIC's DRAMs was eliminated altogether by obtaining a DRAM image from the initial pure C++ reference run, then directly writing this data into the DRAM models from the top-level Verilog testbench. Finally, the Tensilica processor boot time was shortened by removing the slowest portions of the reset vector code, which consisted of loops used to initialize the cache tags and data memory. Again, the top-level Verilog testbench was modified to directly initialize these structures.

Altogether, these accelerations reduced the mixed-level simulation initialization time from three days to about four hours, the vast majority of which consists of Tensilica processor boot time (the accelerated MMR writes take 14 minutes to complete, and the direct DRAM initialization takes seconds). This is still a considerable amount of time, but it is only 40% of the total run time required to compute a single MD time step within a mixed-level simulation using an RTL ASIC model[2], so additional accelerations would not significantly increase the overall test throughput.

[2] The total time required to compute a single time step (~10 hours) is much larger than 1% of the time required to compute 100 time steps (~2.5 days) for two reasons. First, there is a considerable amount of setup code that must run to prepare the computation, independent of the number of time steps. Second, the first time step takes much longer than subsequent time steps due to instruction cache misses.

E. *The Trouble with X's*

One of the obstacles that we faced when running full-ASIC RTL model simulations was dealing with unknown values (X's) in a 4-state (0/1/X/Z) simulator. When a simulation begins there are a large number of uninitialized data values in memories and in registers with unspecified reset behavior. So long as these X's remain confined to datapaths, the simulation can make forward progress, but there are two scenarios in which the X's leak into the ASIC's control logic, rapidly bringing the simulation to a halt. First, if an embedded processor branches conditionally on uninitialized data or attempts to access memory using an undefined address, then the processor state becomes undefined. Second, if a hardware parity check is performed on uninitialized data, then the error detection logic will enter an undefined state, which can cause the entire ASIC model to stop functioning.

In some cases, these X's do not represent an actual error. Uninitialized data is often read from DRAM without being used: this happens, for example, the first time a cache line is read from memory, or when the granularity of a memory read is coarser than the granularity of the valid data being read. Also, software round-robin counters do not need to be initialized in order to function correctly. In other cases, however, the X's are indicative of genuine hardware or software errors, and provide a powerful mechanism for detecting these errors that would otherwise be difficult to isolate. Many of the errors that manifested as X's were reset problems that left various registers in undefined states; others were software bugs where a value was used without being defined.

Because X's were valuable from a verification perspective, we did not want to suppress them by using 2-state simulations. To a large extent we were able to eliminate the software sources of X's by compiling the embedded code for an x86 and running it under *valgrind* [16], which detects the use of uninitialized values. In particular, we required all round-robin counters to be properly initialized. We were also able to ensure that the external host processor would only read valid data from the ASIC model, thus preventing X's from crossing the host interface. Finally, we disabled all parity checking within the on-chip communication ring, allowing the simulation to tolerate packets containing unused, undefined bits. With these adjustments, we were able to make effective use of 4-state RTL simulations for full-ASIC verification.

IV. Numerical Simulation and Verification

Although molecular dynamics is a mature field with well-understood algorithms, existing practice is almost entirely confined to calculations with single- and double-precision floating-point numbers. Anton, on the other hand, uses fixed-point arithmetic to make more effective use of silicon. Anton also uses much more compact lookup tables to evaluate complicated mathematical functions than software running on modern processors, which can quickly and easily access tables of almost arbitrary size. Finally, Anton's rounding rules differ slightly from the "round to nearest" rule employed by IEEE floating point implementations, and we have nowhere near the depth of practical experience with them that we have with IEEE floating point.

These numerical modifications do not qualitatively affect the results of individual time steps, but the algorithms used to run MD computations for billions of time steps are remarkably fragile: seemingly innocuous changes can render a computation unstable in subtle and surprising ways. As a result, in order to be confident that Anton's deviations from common practice will not affect the quality of its output, it was necessary to simulate Anton's behavior over a long enough period of time—millions of time steps—to let any potential interactions between Anton's numerics and the MD algorithms manifest themselves. A common measure of stability, often used as a diagnostic for software MD packages, is conservation of energy. Theoretically, with infinite-precision arithmetic and infinitesimally small time steps, the energy contained in the physical system being modeled ought to remain constant. In practice, energy is only approximately conserved, and the rate at which it changes over time, i.e., the "energy drift", is a measure of the numerical quality of an MD computation.

Millions of time steps are completely beyond the scope of Archsim simulations. A ten million time step run of a small (~13,000 atoms) molecular system requires around 30 billion machine cycles, which represents roughly 100 years of Archsim runtime. We addressed this simulation performance gap by modifying Desmond [17], a highly scalable, parallel software implementation of MD. We reused Desmond's parallel engine, which allows it to run efficiently on hundreds of processors, but re-implemented its floating point computational pipelines to exactly match the fixed-point calculations of Anton. We called the resulting program "Pyrite" to indicate that it is not quite a golden model for the eventual machine because the MD software is subject to change; it simply demonstrates that Anton's numerics are capable of producing stable MD computations.

With Pyrite, we were able to compute almost ten million time steps in about a week for one molecular system; several other systems were run for around one million time steps each. In all cases, the measured energy drift was well within acceptable limits. We were also able to use these long runs to derive statistical properties of the molecular systems that match results based on floating-point MD codes. These runs provided strong reassurance that the numerical choices made in Anton's design do not have adverse numerical consequences. Finally, we reconciled Pyrite's output against that of Archsim across many shorter (tens of time steps) MD computations, ensuring that they both produce bitwise identical results, which completed the top level of our verification chain.

Table 1. Longest MD runs in simulation, measured in total number of time steps, and projected longest run on Anton.

Platform	**Longest run (time steps)**
Archsim (with 1 RTL ASIC)	100
Pyrite	10,000,000
Anton (projected)	400,000,000,000

V. Discussion

Our hierarchical simulation-based design verification strategy for Anton provided a continuous verification chain from MD to logic gates. With Pyrite, we obtained convincing evidence that Anton's customized numerical formats and hardwired datapaths should perform scientifically valid MD computations over billions of time steps. Comparisons between Pyrite and Archsim ensured that Anton is able to produce exactly the same results, bit for bit, as Pyrite. This provided verification at the architectural level, demonstrating the correctness of MD computations on Anton's specialized hardware. Mixed-level co-simulations provided RTL model verification at the ASIC and block levels, ensuring that the Anton ASIC exactly implements the functionality modeled by Archsim. Finally, synthesized logic gates were formally verified against the RTL model, ensuring that these gates would correctly implement the MD calculations.

Anton is particularly amenable to hierarchical verification because its target application, MD, is divided into many discrete time steps, each of which performs exactly the same computational tasks. To establish equivalence between adjacent levels of the hierarchy, it therefore suffices to show that they compute bitwise-identical results for individual time steps. The longest Pyrite run contained five orders of magnitude more time steps than the longest Archsim run (Table 1), but since Pyrite time steps were extensively validated against Archsim time steps at the bit level, we have a high degree of confidence that, were it possible to run 10^7 time steps on Archsim, the results would exactly match those of Pyrite. It is projected that the longest Anton run will contain yet another four orders of magnitude more time steps—a scale of MD computation that has never before been achieved. Ultimately, the only way to confirm that computations of this scale will produce valid scientific results on Anton is to run them on the actual hardware, but our verification hierarchy at least provides compelling evidence that Anton's MD computations should be correct and numerically stable.

The RTL portion of Anton's design verification effort relied heavily on interface binding to effectively remove the boundary between C++ and Verilog. By automatically connecting C++ components to Verilog modules at the interface level and transparently marshalling data between the C++ and Verilog domains during simulation, interface binding makes it possible to implement mixed-level co-simulations with little programmer effort and minimal run-time overhead. This feature of our simulation infrastructure allowed us to reuse Archsim for both block-level and ASIC-level RTL verification environments.

The first Anton ASICs were delivered at the start of 2008. After seven months of testing, during which billions of MD time steps have been run on silicon, six logic bugs have been uncovered. Three of these bugs are located in peripheral logic—tightly coupled to the high-speed controllers for the off-chip DRAM and the communication channels—that was omitted from the full-chip mixed-level environment and was verified separately. The other three involve corner cases related to a numerical overflow signal that were simply never observed during the hundreds of thousands of time steps that were run in simulation on the RTL ASIC model. The natural response of a design verification engineer to these bugs is to question what went wrong, and how we could have found them during the pre-silicon phase. Looking at the bigger picture, however, we have simple software workarounds in place, and are currently performing multi-million time step MD computations on first silicon.

ACKNOWLEDGEMENTS

We would like to thank Joe Bank, Ken Mackenzie and Bill Vick for their valuable contributions to this work, and the anonymous reviewers for their helpful suggestions.

REFERENCES

[1] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, K. J. Bowers, J. C. Chao, M. P. Eastwood, J. Gagliardo, J.P. Grossman, C. R. Ho, D. J. Ierardi, I. Kolossváry, J. L. Klepeis, T. Layman, C. McLeavey, M. A. Moraes, R. Mueller, E. C. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles and S. C. Wang, "Anton: A special-purpose machine for molecular dynamics simulation", in *Proc. 34th International Symposium on Computer Architecture* (ISCA'07), San Diego, CA, June 9–13, 2007, pp. 1–12.

[2] B. Khailany, W. J. Dally, A. Chang, U. J. Kapasi, J. Namkoong and B. Towles, "VLSI design and verification of the Imagine processor", in *Proc. IEEE International Conference on Computer Design* (ICCD'02), Freiburg, Germany, Sept. 16–18, 2002, pp. 289–294.

[3] K. Shimizu, S. Gupta, T. Koyama, T. Omizo, J. Abdulhafiz, L. McConville and T. Swanson, "Verification of the Cell Broadband Engine^{TM} processor", in *Proc. 43rd Design Automation Conference* (DAC'06), San Francisco, California, July 24–28, 2006, pp. 338–343.

[4] J. Walter, "Functional verification of the IBM zSeries eServer z900 system", in *Proc. IEEE International Conference on Computer Design* (ICCD'02), Freiburg, Germany, Sept. 16–18, 2002, pp. 17–21.

[5] P. Gerin, S. Yoo, G. Nicolescu and A. A. Jerraya, "Scalable and flexible cosimulation of SoC designs with heterogeneous multi-processor target architectures", in *Proc. Asia and South Pacific Design Automation Conference* (ASP-DAC'01), Yokohama, Japan, Jan. 30–Feb. 2, 2001, pp. 63–68.

[6] J. Noguera, L. Baldez, N. Simon and L. Abello, "Software-friendly HW/SW co-simulation: An industrial case study", in *Proc. Design, Automation and Test in Europe* (DATE'06), Munich, Germany, March 6–10, 2006, pp. 1–6.

[7] S. Honda, T. Wakabayashi, H. Tomiyama and H. Takada, "RTOS-centric hardware/software cosimulator for embedded system design", in *Proc. International Conference on Hardware/Software Codesign and System Synthesis* (CODES+ISSS'04), Stockholm, Sweden, Sept. 8–10, 2004, pp. 158–163.

[8] A. Hoffmann, T. Kogel and H. Meyr, "A framework for fast hardware-software co-simulation", in *Proc. Design, Automation and Test in Europe* (DATE'01), Munich, Germany, March 13–16, 2001, pp. 760–764.

[9] Open SystemC Initiative. http://www.systemc.org

[10] L. Séméria and A. Ghosh, "Methodology for hardware/software co-verification in C/C++", in *Proc. Asia and South Pacific Design Automation Conference* (ASP-DAC'00), Yokohama, Japan, Jan. 25–28, 2000, pp. 405–408.

[11] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi and M. Poncino, "Legacy SystemC co-simulation of multi-processor systems-on-chip", in *Proc. IEEE International Conference on Computer Design* (ICCD'02), Freiburg, Germany, Sept. 16–18, 2002, pp. 494–499.

[12] T. Kogel, A. Wieferink, R. Leupers, G. Ascheid, H. Meyr, D. Bussaglia and M. Ariyamparambath, "Virtual architecture mapping: A SystemC based methodology for architectural exploration of system-on-chip designs", in *Proc. International Workshop on Systems, Architectures, Modeling and Simulation* (SAMOS'03), Samos, Greece, July 21–23, 2003, pp. 138–148.

[13] R. H. Larson, J. K. Salmon, R. O. Dror, M. M. Deneroff, R. C. Young, J.P. Grossman, Y. Shan, J. L. Klepeis and D. E. Shaw, "High-throughput pairwise point interactions in Anton, a specialized machine for molecular dynamics simulation", in *Proc. 14th International Symposium on High-Performance Computer Architecture* (HPCA'08), Salt Lake City, UT, Feb. 16–20, 2008, pp. 331–342.

[14] J. S. Kuskin, C. Young, J.P. Grossman, B. Batson, M. M. Deneroff, R. O. Dror and D. E. Shaw, "Incorporating flexibility in Anton, a specialized machine for molecular dynamics simulation", in *Proc. 14th International Symposium on High-Performance Computer Architecture* (HPCA'08), Salt Lake City, UT, Feb. 16–20, 2008, pp. 343–354.

[15] Tensilica Inc. http://www.tensilica.com

[16] J. Seward and N. Nethercote, "Using Valgrind to detect undefined value errors with bit-precision", in *Proc. USENIX Annual Technical Conference* (USENIX'05), Anaheim, CA, April 10–15, 2005, pp. 17–30.

[17] K. J. Bowers, E. Chow, H. Xu, R. O. Dror, M. P. Eastwood, B. A. Gregersen, J. L. Klepeis, I. Kolossváry, M. A. Moraes, F. D. Sacerdoti, J. K. Salmon, Y. Shan and D. E. Shaw, "Scalable algorithms for molecular dynamics simulations on commodity clusters", in *Proc. ACM/IEEE Conference on Supercomputing* (SC'06), Tampa, FL, Nov. 11–17, 2006.