Simulation and Embedded Software Development for Anton, a Parallel Machine with Heterogeneous Multicore ASICs

J.P. Grossman, Cliff Young, Joseph A. Bank[1], Kenneth Mackenzie[1],
Douglas J. Ierardi, John K. Salmon, Ron O. Dror, David E. Shaw[2]

D. E. Shaw Research, New York, NY 10036, USA

ABSTRACT

Anton, a special-purpose parallel machine currently under construction, is the result of a significant hardware-software codesign effort that relied heavily on an architectural simulator. One of this simulator's many important roles is to support the development of *embedded software* (software that runs on Anton's ASICs), which is challenging for several reasons. First, the Anton ASIC is a heterogeneous multicore system-on-a-chip, with three types of embedded cores tightly coupled to special-purpose hardware units. Second, a standard 512-ASIC configuration contains a total of 6,656 distinct embedded cores, all of which must be explicitly modeled within the simulator. Third, a portion of the embedded software is dynamically generated at simulation time.

This paper discusses the various ways in which the Anton simulator addresses these challenges. We use a hardware abstraction layer that allows embedded software source code to be compiled without modification for either the simulation host or the hardware target. We report on the effectiveness of embedding golden-model testbenches within the simulator to verify embedded software as it runs. We also describe our hardware-software co-simulation strategy for dynamically generated embedded software. Finally, we use a methodology that we refer to as *concurrent mixed-level simulation* to model embedded cores within massively parallel systems. These techniques allow the Anton simulator to serve as an efficient platform for embedded software development.

Categories and Subject Descriptors

I.6.5 [**Simulation and Modeling**]: Model Development—*Modeling methodologies*

General Terms

Performance, Design, Languages

Keywords

Anton, Simulation, Embedded Software, Special-Purpose Hardware

[1]Joseph A. Bank and Kenneth Mackenzie are also with Reservoir Labs, Inc., New York, NY 10012.

[2]David E. Shaw is also with the Center for Computational Biology and Bioinformatics, Columbia University, New York, NY 10032. E-mail correspondence: David.Shaw@DEShawResearch.com

1. INTRODUCTION

Anton is a special-purpose parallel machine, currently under construction, that is expected to dramatically accelerate molecular dynamics (MD) computations relative to other parallel solutions [13]. The standard Anton configuration will consist of 512 identical ASICs connected by high-speed links to form a three-dimensional torus. Each ASIC contains two computational subsystems: a *high-throughput interaction subsystem* (HTIS) [8], in which specialized datapaths compute pairwise interactions between particles, and a *flexible subsystem* [7], in which programmable embedded processors are used to perform other numerical tasks and to control the overall computation. In total, each ASIC contains 13 embedded processors: an *interaction control block* (ICB) core used to control the HTIS, four general-purpose (GP) cores used to control the flexible subsystem and drive the overall MD algorithm, and eight *geometry cores* (GCs) which perform the bulk of the numerical computation within the flexible subsystem. The GP and ICB cores are specialized versions of the Tensilica LX customizable processor [15], while the GCs were developed specifically for Anton.

Anton's hardware-software codesign effort made extensive use of an architectural simulator to simultaneously evaluate multiple algorithms, hardware accelerators, and software implementations. Initially, this simulator was used for high-level architectural exploration, with the flexible subsystem tasks implemented behaviorally. As the design evolved, functionality was partitioned between hardware blocks and embedded software, and the simulator became a platform for architectural validation, detailed performance estimates, performance/area trade-offs, and embedded software development. In addition, enough detail was added to the simulator to support hardware design verification and the development of system-level control/debug software that runs externally to the ASICs. To our knowledge, this diversity of roles for a single simulator code base is extremely rare in a hardware project of this scope. In this paper, we focus on the mechanisms used to support hardware-software co-simulation and embedded software development within the Anton simulator.

The nature of Anton's embedded software does not lend itself to conventional simulation techniques. The code for the GP cores, which emerged directly from Anton's hardware-software codesign effort, rapidly evolved along with the hardware design, and required strong debugging support from the simulator. There is no compiler for the GCs, so the numerical code exists in two forms: C++ code that runs within a functional GC model, and manually generated assembly code that runs on the GC Instruction Set Simulator (ISS). The ICB code, which delivers a sequence of control instructions to the HTIS, does not even exist at compile time: rather, it is dynamically generated based on the specific molecular system being modeled.

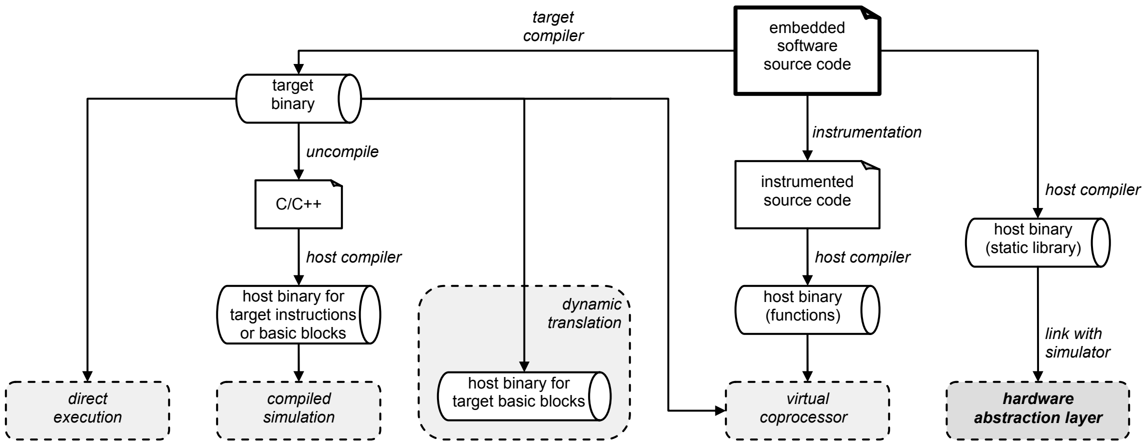

Figure 1. Overview of techniques for accelerating embedded processor simulations. The embedded processor is the *target*; the processor that runs the simulation is the *host*.

Architectural simulations of Anton also differ from traditional embedded processor simulations in both duration and size. A single MD *time step* (the unit of discretization for an MD computation) takes a few microseconds on Anton or, equivalently, a few thousand clock cycles. While Anton is intended to run calculations for hundreds of billions of such time steps (over a quadrillion clock cycles), it generally suffices to simulate fewer than ten time steps for performance modeling and software debugging purposes. On the other hand, simulating a 512-node Anton machine requires significant resources, as such a machine contains a total of 6,656 distinct embedded cores. Thus, architectural simulations of Anton are short (few cycles) but large (many cores, hardware units, and memories) compared to most embedded-processor simulations.

The following sections present our simulation methodologies for Anton's embedded cores; the techniques that we describe address our unique simulation needs and provide an efficient platform for embedded software development. For the GP cores, we show how the use of a hardware abstraction layer allows the full GP embedded software to be compiled without modification for the simulation host processor and linked to the simulator executable. For the GCs, we borrow from hardware design verification by using *golden models* within the simulator. For the ICB cores, we generate a high-level command sequence that can either be directly interpreted or used to produce compilable source code. Finally, we use a methodology, which we refer to as *concurrent mixed-level simulation*, to accelerate ISS-level simulations of massively parallel systems by restricting the use of detailed ISS models to a single simulated node; this technique allows us to efficiently meet our verification and performance estimation requirements.

2. MODELING THE GP CORES USING A HARDWARE ABSTRACTION LAYER

Initial versions of the embedded software were written in C++ and directly linked to the simulator executable. Once we selected an implementation for the GP cores, specifically Tensilica LX processors with customized extensions, it was necessary to adapt the embedded software to the Tensilica processor and modify the simulator to support embedded software development for this platform. The Tensilica processors come with an interpreter-based, cycle-accurate ISS and a mature optimizing C compiler. However, we were concerned about both the speed and memory footprint of the interpreted ISS, especially given that a full-machine simulation would involve 2,048 GP cores.

A variety of clever techniques for accelerating the simulation of embedded processors already exist and offer various trade-offs among speed, accuracy and flexibility (Figure 1). *Direct execution* [12] runs portions of the application software on the simulation host processor rather than on an ISS for the target architecture; this requires the host and target architectures to be the same. When the host and target architectures differ, two methods can be used to accelerate the interpretation of target binaries: *compiled simulation* [1, 11, 17, 20, 21] uncompiles target binaries to a high-level language (usually C or C++) which is then recompiled for the host architecture; *dynamic translation* [9, 10, 18] translates target machine instructions into one or more host machine instructions at simulation time. If the embedded software source code is available, it is possible to adopt the recently proposed *virtual coprocessor* [3] approach in which the source code is automatically instrumented and then compiled for the host architecture. This instrumentation allows the simulation to dynamically switch between the target ISS and the native host binary at function-call boundaries, giving the effect of fast "coprocessor" function calls from the slower ISS.

Although these techniques provide enormous speedups over an interpreted ISS and are extremely flexible in the types of embedded software that they support, they require different workflows for the hardware simulator and the embedded software, which in our case was undesirable. To best support Anton's hardware-software codesign effort, we required a uniform development and debugging environment for both the hardware simulation and the embedded software. We therefore introduced a hardware abstraction layer (HAL) that allows the unmodified GP embedded software to be compiled in its entirety for either the host or target processors. Two corresponding GP core models exist within the simulator: a *high-level model* that runs the embedded software as a native binary, and a *low-level model* that runs the embedded software as a target binary on the Tensilica ISS. This method provides the required uniform development environment as well as extremely fast execution of the embedded software.

The use of a HAL requires defining a hardware-level API with two architecture-specific implementations: a host implementation that runs within the simulator itself, and a target implementation that runs on the target ISS and reads/writes the target processor interface signals modeled by the ISS. All of the hardware functionality available to application programs must be encapsulated within this API, and all embedded software must use the API instead of directly accessing hardware features. This restriction makes the use of a HAL unsuitable for simulation platforms that need to support externally supplied or pre-existing software, unless the cost of refactoring the software to use the API is sufficiently low. In our case, all embedded software was internally developed during the course of the project, so this was not an issue.

2.1 Hardware API

The Anton simulator uses an internally developed cycle-based C++ simulation infrastructure that supports hardware interfaces with ports and connections. We defined a single hardware interface shared by both the high-level (native binary linked with simulator) and low-level (target binary running on ISS) GP core models, so that the choice of model is transparent to the rest of the simulation. The interface consists of several hardware queues, which are used for direct communication between the GP cores, and load/store access to a *remote access unit* (RAU), which performs autonomous data transfers. The hardware API used by the embedded software contains functions for accessing the queues and managing the RAU.

In the high-level GP core model, the embedded software runs natively within a QuickThread [5]—a user-level thread that must be manually scheduled by the simulator, and that must explicitly relinquish control ("*yield*") to suspend its execution and allow the simulator to resume. There is no inherent measure of time in these threads; instead, annotations are added to the embedded software by programmers to model the passage of time. These annotations simply increment the local time and do not otherwise interfere with execution, so the threads are allowed to "run ahead" of the global simulation time. The high-level implementation of each hardware API function first synchronizes to the global simulation time by yielding, if necessary, until the rest of the simulation has caught up. Once the local and global times have been synchronized, the API function can safely execute by reading/writing the appropriate hardware interface ports.

The low-level GP core model uses Tensilica's XTMP cycle-accurate ISS [16], which advances in lockstep with the main cycle-driven simulation. The hardware API is implemented in C; the queue access functions (push, pop, peek) delegate to corresponding processor instructions, while the RAU management functions perform reads and writes to a memory-mapped RAU interface. When an API function reads or writes a processor interface signal, the XTMP library invokes a callback function within the simulator that forwards the read/write to the appropriate port(s) of the hardware interface (Figure 2).

2.2 Memory Visibility

Each GP core in the flexible subsystem has a private data cache, and is connected to an SRAM that is shared with the RAU and two of the GCs. For both GP core models, we needed to preserve two types of visibility into these memories. First, *data visibility*

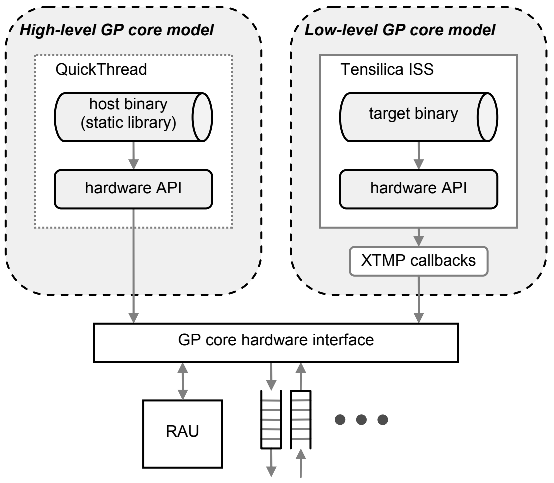

Figure 2. The high-level and low-level GP core models share a common GP core hardware interface. The interface is accessed directly by the high-level hardware API, and indirectly by the low-level hardware API via XTMP callbacks.

was required for various global validation functions. At certain points in the MD time step, these functions check the consistency of data structures across the machine using "simulator magic", i.e. by directly accessing deeply buried implementation state. These consistency checks were critical for debugging the embedded software, and required visibility into the contents of both the data cache and the shared SRAM. Second, *access visibility* was required for accurate performance modeling. SRAM bandwidth and contention were important factors contributing to performance, so the simulator needed to be able to detect SRAM accesses to properly account for these effects.

Data visibility was fairly straightforward to implement. The shared SRAM was modeled independently and was always available for inspection. In the low-level GP core model, the data cache is, by default, *not* visible because it is part of the ISS, but we restored visibility by setting the cache mode to write-through. We used the same in-memory data structure (defined by a C struct) to hold the contents of the cache for both the high-level and low-level GP core models: in the high-level model the embedded software modified this structure directly, while in the low-level model this structure was a copy of the contents of the ISS cache, and was kept up-to-date by the XTMP write-through callback handler. A small (but important) note is that some care was required in aligning the contents of this data structure to ensure that both the host compiler (gcc) and the Tensilica compiler (xt-xcc) would produce exactly the same data layouts.

Obtaining visibility to SRAM accesses required slightly more engineering. This visibility was already present in the low-level model because XTMP uses callbacks to access SRAM, but in the natural high-level implementation, the embedded software would directly read and write the SRAM data structure elements, so there would be no way for the simulator to monitor these accesses. Our solution was to wrap every reference to SRAM within a macro. In the target builds (which run on the ISS), this macro expands to a direct data access. In the host builds (which are linked to the simulator), the macro expansion also includes a notification callback, which allows the simulator to properly model SRAM contention between the GP cores and the other hardware components.

2.3 Instrumentation

One of the advantages of a HAL-based methodology is that it makes it easy to add various instrumentation to the code (assertions, validation, printfs, etc.) by using macros that are only implemented in the host builds. This approach is not possible with simulation techniques that use target binaries as an intermediate representation, although we note that the virtual coprocessor technique not only supports instrumentation, but can in some cases add it to the code automatically.

The instrumentation functions use simulator magic to perform their duties and cannot be implemented in the production binaries that run on the Anton ASIC. It was, however, desirable to preserve their functionality in the ISS-level simulations. We accomplished this by defining three versions of the instrumentation macros depending on preprocessor definitions: in the host builds, the macros become function calls; in production target builds intended for the Anton ASIC, they are simply removed; and in target builds intended for ISS simulation, they are implemented as writes to an otherwise-unused portion of data memory. For these "instrumentation writes", the memory address specifies the instrumentation function to invoke, and the 128-bit write data contains up to four arguments. The XTMP callback that handles data memory writes then invokes the appropriate instrumentation function with the supplied arguments.

3. VERIFYING GC CODE WITH EMBEDDED GOLDEN MODELS

We did not have a compiler for the GCs, which placed us in the unenviable position of having to write two versions of the numerical code: a high-level version, written in C++, which could be linked to the simulator executable; and a low-level version, written in assembly, which would run on the ISS. In our software development workflow, the high-level code has always been written and tested first for two reasons. First, it is much easier and faster to work with natively-compiled C++ than emulated assembly. Second, C++ versions of the numerical computations are required by the *sequential validator*—a single-threaded implementation of the MD algorithm used to verify (bitwise) the computation performed on the simulated parallel machine. Once the high-level GC code has been implemented and tested, the identical computation is then hand-coded in assembly.

We considered a variety of approaches for working within a single code base, either by programming entirely at the assembly level and using compiled simulation techniques to address simulation speed, or by using a stylized subset of C that could be automatically converted to efficient assembly language without the need for a full-featured compiler. None of these approaches, however, would have provided us with both the ease of coding and sequential validator support that we desired for initial development, and the highly optimized assembly code that we required for production binaries.

Because we were unable to eliminate the dual implementations of the GC code, we strove instead to leverage these multiple implementations to our advantage. We did so by making use of the standard *golden model* methodology for block-level hardware design-verification testbenches. In this methodology, the device under test (DUT) is simulated side-by-side with a *golden model*—a behavioral implementation of the block that conforms to the hardware specification. The DUT and the golden model are presented with identical stimuli, and the testbench compares their outputs. In the Anton simulator, each GC ISS (which acts as the DUT) is instantiated alongside a high-level GC (which acts as the golden model) within an "embedded testbench". The simulator supplies the same inputs to each model, and verifies that the outputs are identical. In this manner, we obtain substantial verification of the GC assembly code by ensuring that its results exactly match those produced by the reference C++ implementation.

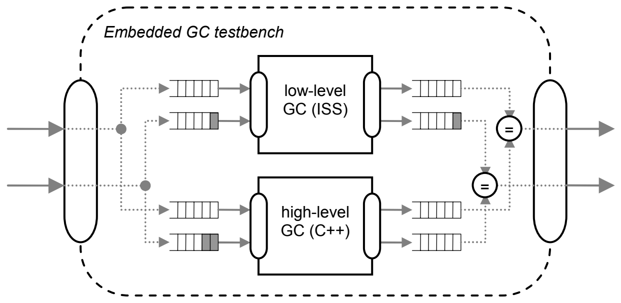

Figure 3. The embedded GC testbench has the same hardware interface as a single GC, but contains both high-level and low-level GC models. Input data is copied to both models. Output data is synchronized and compared before being forwarded to the rest of the simulation.

This technique proved to be extremely effective for isolating bugs in the numerical code. Although these bugs would eventually manifest as miscomparisons between the outputs of the ISS-level Anton simulator and the sequential validator, such miscomparisons are difficult to diagnose because the source of the problem could be in any portion of the MD computation. Using golden model verification for the GCs localized these bugs both in *origin* (which GC was the source of the error) and in *time* (at which point of the MD computation the error occurred), enormously simplifying the debugging task.

3.1 Transaction-Level Verification

The high-level GC models, like the high-level models of the GP cores, execute natively within QuickThreads and have no intrinsic notion of time. The code is manually annotated with the amount of time that various computations take, based on performance numbers obtained from the low-level (ISS) simulations. The resulting timing of the high-level model is an approximation only; in particular, it is not possible to compare the outputs of the high- and low-level GC models on a cycle-by-cycle basis. Instead, golden model verification is performed at the transaction level, which is made significantly easier by the fact that the GC hardware interface is entirely queue-based. Input data is copied into two corresponding queues, one for each GC model. Output data from the two models is merged: when data is available on the corresponding output queues in both models, it is compared bitwise before being forwarded to the rest of the simulation. This functionality is encapsulated within a C++ class that contains both GC models and has the same hardware interface as a single GC, so that the instantiation of a verification testbench in place of a GC is transparent to the rest of the simulation (Figure 3).

4. SIMULATING THE ICB USING COMMAND SEQUENCE INTERPRETATION

The ICB core in the HTIS is responsible for coordinating the movement of data from a set of memory buffers to an array of hardware datapaths that compute pairwise particle interactions. It does so by pushing a sequence of buffer allocation, data movement and synchronization commands onto a queue; the commands are then executed by the HTIS hardware. This command-queue interface provides a convenient abstraction layer for the ICB embedded software: within the simulator, an ICB model emits a sequence of commands that are directly interpreted by the HTIS hardware model. The specific sequence of commands required to orchestrate the HTIS computation, however, depends on both the size of the molecular system being modeled and the parameters of the MD computation, and as such is not known at compile time. Instead, these command sequences are dynamically generated by the simulator executable as a preparation step; the high-level ICB model simply stores the generated commands and pushes them onto the command queue.

Once the Tensilica ICB core was specified, including specialized processor instructions used to push commands onto the queue, actual binaries were required for the detailed ISS simulations. A generic binary would have been too large to fit in the ICB core's limited instruction memory; it was therefore necessary to dynamically generate binaries specialized to the molecular system being modeled and the parameters of the MD computation. This was accomplished by expanding the preparation step to automatically generate compilable C code from the internal representation of the command sequence. Each command is emitted as a single macro or processor instruction; a fixed header file provides definitions for all required macros, constants, and inline functions. The Tensilica compiler is invoked from within the simulator executable to generate the ICB binary, which is then interpreted by the Tensilica ISS during the simulation (Figure 4).

5. CONCURRENT MIXED-LEVEL SIMULATION

High-level processor models are much faster than ISS-level models, and are therefore generally preferred for embedded software development. It is, however, still necessary to test the embedded software on ISS-level models, as this is the only way to ensure the correctness of the target executable. One of the problems we encountered with the GP code, for example, was related to an unanticipated reordering of memory accesses by Tensilica's optimizing compiler. The ISS models are also important for verifying the interaction between the processors and the rest of the system; several problems with the ICB core were discovered when the ISS-level ICB model failed to exactly reproduce the original sequence of HTIS commands. A second function of ISS-level simulations is to provide more accurate performance estimates, particularly when timing annotations in the embedded software are missing or outdated.

One of the simplest and most commonly used techniques to address the slow speed of ISS-level simulation is "fast-forwarding", which uses fast high-level models to initialize the simulation, then switches to slower detailed models to obtain performance estimates for targeted regions of code [14, 19]. This approach is effective for conventional simulations in which the run time for

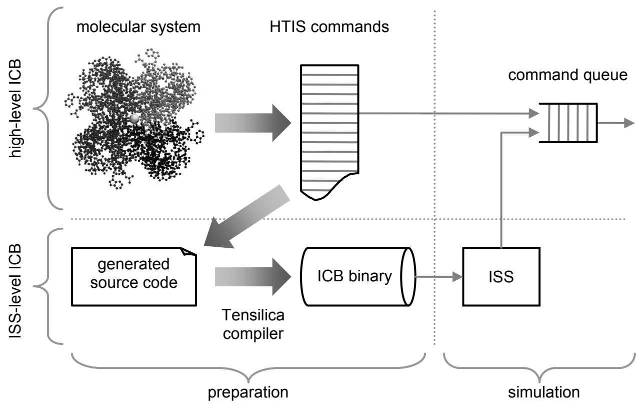

Figure 4. The high-level ICB model pushes a list of generated HTIS commands directly onto the command queue. For the ISS-level ICB model, the command sequence is used to generate and compile source code which is executed on the ICB core ISS; the ISS regenerates the original commands and places them on the command queue.

the software of interest is much shorter than the processor initialization time, but for Anton the reverse is true—the majority of time is spent performing the computations for which we desire ISS-level validation and performance estimates.

ISS-level simulations of Anton are also challenging due to the large number of embedded processors in a 512-node machine. In fact, the XTMP ISS supplied by Tensilica for the GP and ICB cores does not even support simulations of this scale: it is only available as a 32-bit library, requiring the simulator to fit within a 4 GB virtual address space, but XTMP is fairly inefficient in its use of memory and would require in excess of 20 GB to simulate the requisite 2560 Tensilica cores. In principle, one could address this limitation by parallelizing the simulation itself, but we instead used an alternate methodology that altogether eliminates the need for full-machine ISS simulations.

A key observation is that all 512 nodes in a full machine run the same embedded software, so from a verification perspective it suffices to use ISS models for a single node only. We adopted this approach, which we refer to as *concurrent mixed-level simulation* (CMLS) to distinguish it from mixed-level simulations that sequentially alternate between high- and low-level processor models. Because the use of detailed models is restricted to a single node, the overall impact on memory usage and execution time is significantly reduced from a full-machine ISS-level simulation. Mixed-level simulations have been applied to single-chip multiprocessor designs with different processors modeled using different levels of detail [4]. CMLS, in the context of a parallel machine, has the advantage that a full ASIC is modeled in detail, so a single simulation tests all embedded software at the ISS level, and also tests all pairs of interactions between ISS-level processor models.

On its own, CMLS does not give very good performance estimates, so we used a standard annotation approach to obtain more accurate performance data. The first step is to run a simulation with one ISS-level node; the execution traces from this node are used to annotate the embedded software with appropriate delays. High-level simulations can then be run with the annotated software to obtain performance estimates. Table 1 shows the results of this methodology for a 64-node Anton configuration (sized so that a full-ISS simulation would fit in memory). Before annotation, the high-level simulation overestimates performance by 55% compared to the full-ISS simulation. The CMLS is more accurate, but still overestimates performance by 31%. When data from the CMLS run is used to annotate the embedded software, the annotated high-level simulation is accurate to within 15%, but runs nearly seven times faster than the full-ISS simulation. We note that while a manual annotation process was sufficient for our purposes, one can also automatically obtain timing annotations from a static analysis of the target executable [2, 6].

Table 1. Anton performance estimates obtained from four different types of simulation of a 64-node configuration.

Type of Simulation	**Predicted Performance (μs per MD time step)**	**Error**
Full ISS	44.7	*baseline*
High-level (unannotated)	19.9	55%
CMLS	31.0	31%
High-level (annotated)	38.0	15%

6. CONCLUSION

Simulation was central to the hardware-software codesign process that gave rise to the Anton architecture. This architecture, which will allow Anton to achieve dramatic speedups over general-purpose approaches to MD, originally existed as pure software within early versions of the Anton simulator. As the architecture was refined, a single code base was partitioned into a hardware simulation tightly coupled to embedded software. The techniques described in this paper—using a hardware abstraction layer, golden model comparison of high- and low-level GC code, command sequence interpretation, and concurrent mixed-level simulation—allowed the continued use of the Anton simulator as an effective platform for embedded software development. The high fidelity and strong debugging support afforded by these techniques were essential for the pre-silicon development of fully-functional embedded software. As a testament to the success of the simulator in this regard, when the first Anton chips were delivered at the start of 2008, the embedded software—developed entirely within simulation—was ported to the actual hardware in a matter of days with only minor modifications.

7. References

[1] M. Burtscher and I. Ganusov. Automatic synthesis of high-speed processor simulators. *37th International Symposium on Microarchitecture* (MICRO-37), Portland, Oregon, Dec. 4–8, 2004, 55–66.

[2] S. Dwarkadas, J. R. Jump and J. B. Sinclair. Execution-driven simulation of multiprocessors: address and timing analysis. *ACM Trans. on Modeling and Computer Simulation*, 4(4), Oct. 1994, 314–338.

[3] L. Gao, S. Kraemer, R. Leupers, G. Ascheid and H. Meyr. A fast and generic hybrid simulation approach using C virtual machine. *International Conference on Compilers, Architectures, and Synthesis for Embedded Systems* (CASES '07), Salzburg, Austria, Oct. 2007, 3–12.

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

[5] D. Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06. University of Washington Dept. of Computer Science and Engineering, 1993.

[6] S. Kraemer, L. Gao, J. Weinstock, R. Leupers, G. Ascheid and H. Meyr. HySim: a fast simulation framework for embedded software development. *International Conference on Hardware-Software Codesign and System Synthesis* (CODES+ISSS '07), Salzburg, Austria, Sept. 30–Oct. 5, 2007, 75–80.

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

[8] R. H. Larson, J. K. Salmon, R. O. Dror, 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. *14th International Symposium on High-Performance Computer Architecture* (HPCA-14), Salt Lake City, UT, Feb. 16–20, 2008, 331–342.

[9] W. S. Mong and J. Zhu. DynamoSim: a trace-based dynamically compiled instruction set simulator. *2004 IEEE/ACM International Conference on Computer-Aided Design* (ICCAD '04), Washington, D.C., Nov. 7–11, 2004, 131–136.

[10] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr and A. Hoffman. A Universal technique for fast and flexible instruction-set architecture simulation. *39th Conference on Design Automation* (DAC '04), New Orleans, LA, June 10–14, 2002, 22–27.

[11] M. Reshadi, P. Mishra and N. Dutt. Instruction set compiled simulation: a technique for fast and flexible set simulation. *40th Conference on Design Automation* (DAC '03), Anaheim, CA, June 2–6, 2003, 758–763.

[12] M. Rosenblum, S. A. Herrod, E. Witchel and A. Gupta. Complete computer system simulation: the SimOS approach. *IEEE Parallel & Distributed Technology: Systems & Applications*, 3(4), Winter 1995, 34–43.

[13] 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. *34th International Symposium on Computer Architecture* (ISCA '07), San Diego, CA, June 9–13, 2007, 1–12.

[14] P. K. Szwed, D. Marques, R. M. Buels, S. A. McKee and M. Schulz. SimSnap: fast-forwarding via native execution and application-level checkpointing. *Eighth Annual Workshop on Interaction between Compilers and Computer Architectures* (INTERACT '04), Madrid, Spain, Feb. 15, 2004, 65–74.

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

[16] Tensilica, Inc. Xtensa XTMP. http://www.tensilica.com/products/sw_xtmp_xtsc.htm

[17] J. E. Veenstra and R. Fowler. MINT: a front end for efficient simulations of shared-memory multiprocessors. *Second International Workshop on Modeling, Analysis, and Simulation on Computer and Telecommunication Systems* (MASCOTS '94), Durham, NC, Jan. 31–Feb. 2, 1994, 201–207.

[18] E. Witchel and E. Rosenblum. Embra: fast and flexible machine simulation. *ACM SIGMETRICS '96: Conference on Measurement and Modeling of Computer Systems*, Philadelphia, PA, May 23–26, 1996, 68–79.

[19] R. E. Wunderlich, T. F. Wenisch, B. Falsafi and J. C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. *30th International Symposium on Computer Architecture* (ISCA '03), San Diego, CA, June 2003, 84–95.

[20] J. Zhu and D. D. Gajski. A Retargetable, ultra-fast instruction set simulator. *Design, Automation and Test in Europe Conference and Exhibition* (DATE '99), Munich, Germany, March 9–12, 1999, 298–302.

[21] V. Živojnović, S. Tjiang and J. Meyr. Compiled simulation of programmable DSP architectures. *1995 IEEE Workshop on VLSI Signal Processing*, Sakai, Japan, 1995, 27–35.