JiST: EMBEDDING SIMULATION TIME INTO A VIRTUAL MACHINE

Rimon Barr, Zygmunt J. Haas, Robbert van Renesse

Computer Science and Electrical Engineering,
Cornell University, Ithaca NY 14853
{barr@cs, haas@ee, rvr@cs}.cornell.edu

ABSTRACT

In this paper, we propose a new, unifying approach for constructing simulators, called virtual machine-based simulation, that combines the advantages of the traditional systems-based and language-based simulator designs. We introduce JiST, a Java-based simulation platform that embodies this design, executing discrete event simulations efficiently by embedding simulation semantics directly into the Java execution model and transparently performing important simulation optimizations and cross-cutting program transformations at the bytecode level. The system provides standard benefits that the modern Java runtime affords. In addition, JiST is efficient, out-performing existing highly optimized simulation runtimes both in space and time.

KEYWORDS

discrete event simulation, simulation languages and environments, virtual machine, Java

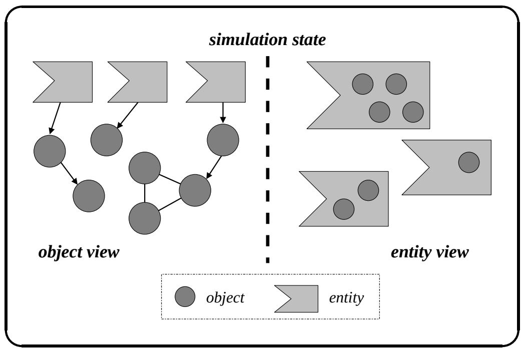
1 INTRODUCTION

Due to their popularity and widespread utility, discrete event simulators have been the subject of much research (surveyed in [8, 10, 6]). From a systems perspective, researchers have built many types of simulation *libraries* or execution *runtimes* spanning the gamut from the conservatively parallel to the optimistic. From a modeling perspective, researchers have designed numerous simulation *languages* that codify event causality and simulation state constraints, simplifying simulation development and permitting important static and dynamic optimizations.

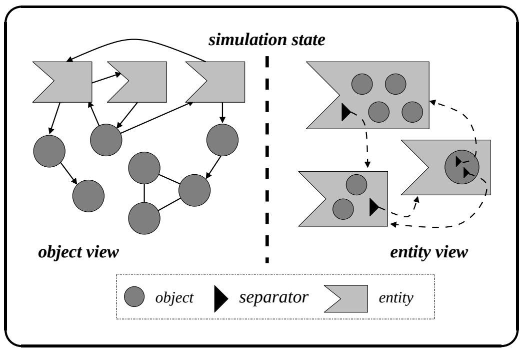
Yet, despite a plethora of ideas and contributions to theory, languages and systems, the parallel simulation community has repeatedly asked itself "will the field survive?" under a perception that it had "failed to make a significant impact in the general simulation community," (see [5, 9, 2] and others). Even though a number of parallel discrete event simulation environments have been shown to scale beyond 10^4 nodes [12], slow sequential simulators remain the norm. These observations influenced the JiST project. Specifically, we decided to:

- *not* invent a simulation language – new languages, and especially domain-specific ones, are rarely adopted by the broader community;
- *not* create a simulation library – libraries often require developers to litter their code with PDES-specific, non-portable library calls and impose unnatural program structure to achieve performance and concurrency; and
- *not* develop a new language runtime – custom runtimes are rarely as optimized, reliable, featured, or portable as their generic counterparts.

Instead, we propose a new approach to building simulators: to bring simulation semantics to a modern and popular virtual machine-based language. JiST, which stands for **J**ava **i**n **S**imulation **T**ime, is a new discrete event simulation system built along these principles. Specifically, the key motivation behind JiST is to create a simulation system that can execute discrete event simulations *efficiently*, yet achieve this *transparently* within a *standard* language, where:



(a)



(b)

Figure 1. (a) Simulation programs are partitioned into entities along object boundaries. Thus, entities do not share any application state and can independently progress through simulation time. (b) At runtime, entity references are transparently replaced with separators, which both preserves the separation of entity state and serves as a convenient point to insert functionality.

- **standard** means writing simulations in a conventional, popular programming language, as opposed to a domain-specific language designed explicitly for simulation;
- **efficient** denotes optimizing the simulation program statically and dynamically by considering simulation state and event causality constraints in addition to general-purpose optimizations; creating a simulation engine that compares favorably with existing, highly optimized systems both in terms of simulation throughput and memory consumption;
- and **transparent** implies the separation of efficiency from *correctness*; that correct simulation programs can be transformed to run efficiently without the insertion of simulation-specific library calls or other manual program modifications. **Correctness** is an assumed pre-condition that simulation programs must compute valid and useful results, regardless of how they are constructed.

These three attributes – the last one in particular – highlight an important distinction between JiST and previous simulation systems in that the simulation code that runs on JiST need not be written in a domain-specific language invented specifically for writing simulations, nor need it be littered with special-purpose system calls and call-backs to support runtime simulation functionality. Instead, JiST transparently introduces simulation time execution semantics to simulation programs written in plain Java and they are executed over an unmodified Java virtual machine. In other words, JiST converts a virtual machine into a simulation system that is flexible and efficient. In the remainder of this paper, we highlight the fundamentals of virtual machine-based simulation. The complete JiST system, along with more detailed and complete explanations, is available online [3].

2 DESIGN AND IMPLEMENTATION

The purpose of the JiST system is to run discrete event simulations both *efficiently* and *transparently* using a *standard* systems language and runtime. In this section, we elaborate very briefly on what it means to execute a program in *simulation time*, and provide an overview of how the JiST system supports this abstraction.

2.1 Simulation programs

JiST simulation programs are written in plain Java, an object-oriented language. Thus, during its execution, the state of the program is contained within individual objects. These objects communicate by passing messages, represented as object method invocations in the language.

JiST extends this traditional programming model with the notion of *entities*. Entities logically encapsulate application objects, depicted in Figure 1a, and serve to demarcate independent simulation components. Every application object is contained within exactly one entity, and simulation events are then intuitively represented as method invocations among these entities, as shown in Figure 1b. This is a convenient abstraction that not only eliminates the need for an explicit simulation event queue, but also enforces a clean partitioning of the simulation state.

JiST manages a simulation at the granularity of its entities. Instructions and method invocations *within* an entity follow the regular Java semantics, entirely opaque to the JiST infrastructure. The vast majority of this code is involved with encoding the logic of the simulation model and is entirely unconnected to the notion of simulation time. All the standard Java class libraries are available and behave as expected. In addition, the simulation developer has access to a few basic JiST primitives, including functions such as `getTime` and `sleep`, which return the current simulation time and advance it, respectively.

In contrast, invocations *across* entities are executed in simulation time. This means that an invocation is performed on the callee entity when its state is at the same simulation time as the calling entity. Thus, cross-entity method invocations act as synchronization points in simulation time. The invocations are non-blocking. They are queued in simulation time order and are performed without continuation. Because of the simulation partitioning, interactions among entities can only occur via the JiST infrastructure.

Since entities do not share any application state, each entity can actually progress through simulation time independently between interactions. Thus, by tracking the simulation time of each entity individually, the model can support concurrent execution. By adding entity checkpointing, the model can even support speculative execution.

2.2 Hello world!

These basic extensions allow us to write simulation programs. The simplest such program, that still uses simulation time semantics, is a counterpart of the obligatory "hello world" program. It is a simulation with only a single entity that emits one message at every simulation time-step, as listed in Figure 2.

This short simulation program highlights some important concepts. To begin, the `hello` class is an entity, since it implements the `JistAPI.Entity` interface (line 2). Entities can be created (line 5) and their methods invoked (lines 6 and 10) just as any regular Java object. The entity method invocation, however, happens in simulation time. This is most apparent on line 10, which is a seemingly infinite recursive call. In fact, if this program is run under a regular Java virtual machine (i.e. without the JiST rewriting machinery) then we would encounter a stack overflow at this point. However, under JiST, the semantics is to schedule the invocation via the simulation time kernel, and thus the call becomes non-blocking. Therefore, the `myEvent` method, when run under JiST semantics, will advance simulation time by one time step (line 9), then schedule a new event at that future time, and finally print a hello message (line 11) with the entity simulation time (line 11). As expected, the output is:

```
simulation start  hello world, t=1 hello world, t=2 etc.
```

2.3 Program transformation

The JiST system consists of four distinct components: a compiler, language runtime or virtual machine, rewriter and simulation time kernel, as shown in Figure 3. The compiler and runtime components of the JiST system can be any standard Java compiler and virtual machine, respectively. Simulation time execution semantics are introduced by the two remaining components. The *rewriter* component of JiST is a dynamic class loader. It intercepts all class load requests, and subsequently verifies and modifies the requested classes. These modified, rewritten classes now incorporate the embedded simulation time operations, but otherwise completely preserve hello.java

```
import jist.runtime.JistAPI;
class hello implements JistAPI.Entity {
  public static void main(String[] args) {
    System.out.print("simulation start ");
    hello h = new hello();
    h.myEvent();
  }
  public void myEvent() {
    JistAPI.sleep(1);
    myEvent();
    System.out.print(" hello world, t="+JistAPI.getTime());
  }
}
```

Figure 2. "Hello world": This simple simulation emits a single message per time step.

the existing program logic. At runtime, the modified classes interact with the *simulation time kernel* through the various injected operations.

The purpose of the rewriter is to transform the JiST instructions embedded within the compiled simulation program into code with the appropriate simulation time semantics. Below, we describe just the fundamental simulation time transformation. Many other orthogonal program transformations and well-known simulation-specific optimizations have been implemented or are possible. The rewriter is implemented using the Byte-Code Engineering Library [4], and its overhead, both in terms of class loading time and increase in class file size, is negligible.

The basic design of the rewriter is that of a multi-pass visitor over the class file structure, traversing and possibly modifying the class, its fields and methods, and their instructions, based on the following set of rules. The rewriter first verifies an application by performing bytecode checks, in addition to the standard Java verifier, that are specific to simulations. Specifically, it ensures that all classes that are tagged as entities conform to entity restrictions: the fields of an entity must be non-public and non-static; all public methods should be concrete and should return `void`; and some other minor conditions. These ensure that the state of an entity is completely restricted to its instance, and also allow entity methods to be invoked without continuation, as per simulation time semantics.

Conforming to the earlier-stated goal of partitioning the application state, entities are never referenced directly by other entities. This isolation is achieved by the insertion of stub objects, called *separators*. The rewriter also adds a self-referencing separator field to each entity and code to initialize it using a unique reference provided by the simulation time kernel upon creation.

For uniformity, all entity field accesses are converted into method invocations. Then, all method invocations on entities are subsequently replaced with invocations to the simulation time kernel. This invocation requires the caller entity time, the method invoked, the target instance and the invocation parameters: the simulation time comes from the kernel; the method invoked is identified using an automatically created and pre-initialized method reflection stub; the target instance is identified using its separator, which is found on the stack in place of the regular Java object reference, along with the invocation parameters, which must be packed into an object array to conform with Java calling conventions. The rewriter injects all the necessary code to do this inline, and also deals with the complications of handling primitive types, the `this` keyword, constructor invocation restrictions, static initializers, and other Java-related details.

The rewriter then modifies all entity creations in all classes to place a separator on the stack in place of the object reference. All entity types in all entities are also converted to separators, namely in: field types, method parameter types, and method return types, as well as typed

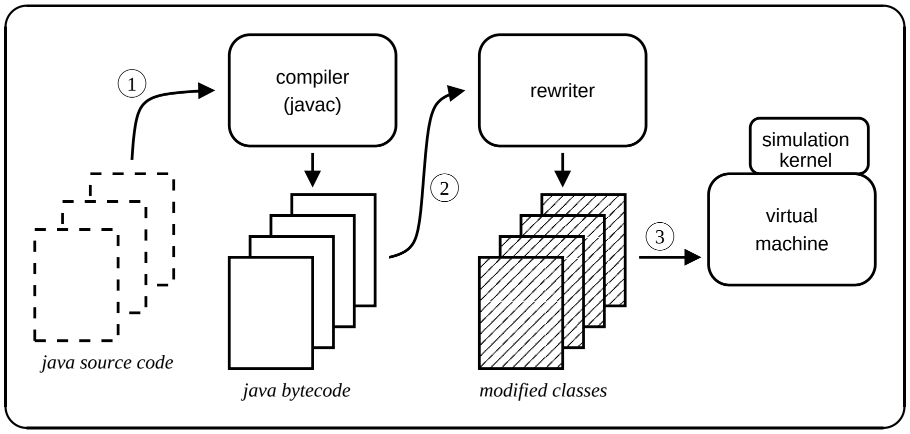


Figure 3. The JiST system architecture – simulations are (1) compiled, then (2) dynamically instrumented by the rewriter and finally (3) executed. The compiler and virtual machine are standard Java language components. Simulation time semantics are introduced by the rewriter, and supported at runtime by the simulation time kernel.

instructions, including field accesses, array accesses and creation, and type casting instructions. Finally, all static calls to the `JistAPI` object are converted into equivalent implementations that invoke functionality of the simulation time kernel.

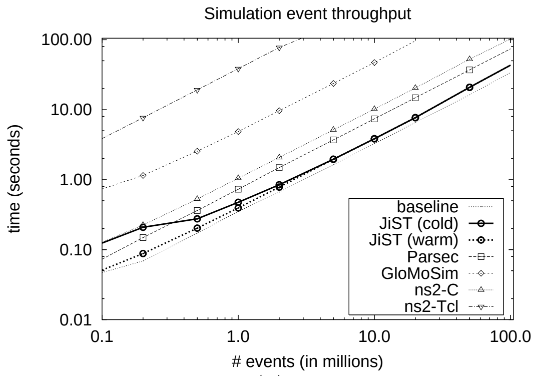
In addition to the entity-related program modifications, the rewriter also performs various static analyses that help drive runtime optimizations. For instance, the rewriter identifies open-world immutable objects [11] and labels them as *timeless*, which means that they may be passed by reference across entities, not by copy. In some cases, the analysis can be overly conservative due to Java static type restrictions. The programmer can then explicitly define an object to be timeless, implying that the object will not be modified in the future even though it technically could be.

3 EVALUATION

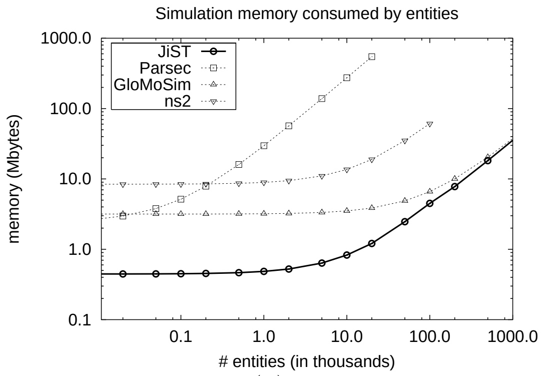
Though conventional wisdom regarding language performance [1] might suggest against implementing our system in Java, we show that JiST and SWANS, a scalable network simulator built atop our platform, perform surprisingly well. We present some JiST micro-benchmarks of event throughput and memory consumption. JiST has twice the raw event throughput of the highly optimized, C-based Parsec [2] engine (Figure 4a). The JiST design also results in very small memory footprints, out-performing all the competing simulator designs (Figure 4b).

Comparing SWANS with the two most popular ad hoc wireless network simulators, ns2 [7] and GloMoSim [13], highlights the practical advantages of JiST under realistic workloads. The results shown are for simulations of a heartbeat node discovery protocol. SWANS has more than ten times the event throughput of ns2, and implements an efficient signal propagation protocol that allows simulation time to scale linearly with the simulated network size (Figure 4c). Within the same memory limits, SWANS can model networks that are one and two orders of magnitude larger than what is possible with GloMoSim and ns2, respectively (Figure 4d).

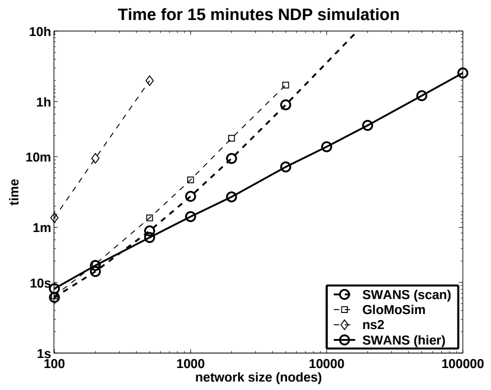
The net result of the various design decisions and optimizations is a wireless network simulator that, as an example, can simulate a million node network running a node discovery protocol within approximately 6 hours on a 2.0 GHz uni-processor with 2 GB of RAM, or well over an order of magnitude beyond existing wireless network simulator capabilities.



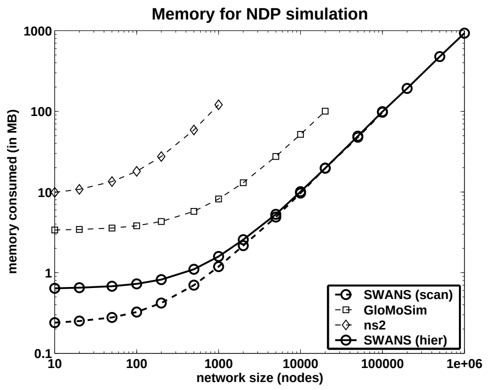
(a)



(b)



(c)



(d)

Figure 4. JiST (a) throughput and (b) memory micro-benchmarks. SWANS (b) throughput and (c) memory macro-benchmarks running simulations of the node discovery protocol. Note the log-log axes.

4 CONCLUSION

We propose and briefly outline a new approach for constructing simulators, called virtual machine-based simulation. JiST is a Java-based simulation engine that embodies this design. It is efficient, out-performing existing highly optimized simulation runtimes, and inherently flexible, capable of transparently performing cross-cutting program transformations and optimizations at the bytecode level. For further details, please refer to software and documentation online [3].

REFERENCES

[1] D. Bagley. The great computer language shoot-out, 2001. `http://www.bagley.org/~doug/shootout/`.

[2] R. L. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, and H. Y. Song. Parsec: A parallel simulation environment for complex systems. *IEEE Computer*, 31(10):77–85, Oct. 1998.

[3] R. Barr and Z. J. Haas. JiST/SWANS website, 2004. `http://www.cs.cornell.edu/barr/repository/jist/`.

[4] M. Dahm. Byte code engineering with the BCEL API. Technical Report B-17-98, Freie Universität Berlin, Institut für Informatik, Apr. 2001.

[5] R. M. Fujimoto. Parallel discrete event simulation: Will the field survive? *ORSA Journal on Computing*, 5(3):213–230, 1993.

[6] R. M. Fujimoto. Parallel and distributed simulation. In *Winter Simulation Conference*, pages 118–125, Dec. 1995.

[7] S. McCanne and S. Floyd. ns (Network Simulator) at `http://www-nrg.ee.lbl.gov/ns`, 1995.

[8] J. Misra. Distributed discrete event simulation. *ACM Computing Surveys*, 18(1):39–65, Mar. 1986.

[9] D. M. Nicol. Parallel discrete event simulation: So who cares? In *Workshop on Parallel and Distributed Simulation*, June 1997.

[10] D. M. Nicol and R. M. Fujimoto. Parallel simulation today. *Annals of Operations Research*, pages 249–285, Dec. 1994.

[11] I. Pechtchanski and V. Sarkar. Immutability specification and its applications. In *Java Grande*, Nov. 2002.

[12] G. Riley and M. Ammar. Simulating large networks: How big is big enough? In *Conference on Grand Challenges for Modeling and Sim.*, Jan. 2002.

[13] X. Zeng, R. L. Bagrodia, and M. Gerla. GloMoSim: a library for parallel simulation of large-scale wireless networks. In *Workshop on Parallel and Distributed Simulation*, May 1998.