SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2004; **00**:1–7 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

JiST: An efficient approach to simulation using virtual machines[†]

Rimon Barr,[1] Zygmunt J. Haas,[2] and Robbert van Renesse[1]

[1] *Department of Computer Science, Cornell University, Ithaca, NY 14853*
[2] *Department of Electrical Engineering, Cornell University, Ithaca, NY 14853*
E-mail: rimon@acm.org, haas@ece.cornell.edu, and rvr@cs.cornell.edu

SUMMARY

Discrete event simulators are important scientific tools and their efficient design and execution is the subject of much research. In this paper, we propose a new approach for constructing simulators that leverages virtual machines and combines advantages from the traditional systems-based and language-based simulator designs. We introduce JiST, a Java-based simulation system that executes discrete event simulations both efficiently and transparently by embedding simulation semantics directly into the Java execution model. The system provides standard benefits that the modern Java runtime affords. In addition, JiST is efficient, out-performing existing highly optimized simulation runtimes. As a case study, we illustrate the practicality of the JiST framework by applying it to the construction of SWANS, a scalable wireless ad hoc network simulator. We simulate million node wireless networks, which represents two orders of magnitude increase in scale over what existing simulators can achieve on equivalent hardware and at the same level of detail.

KEY WORDS: discrete event simulation, simulation languages, Java, wireless networks, aspect-oriented programming

[†]As of this writing, the JiST and SWANS software distributions and documentation are available online [1].
Contract/grant sponsor: This work has been supported in part by the DoD Multidisciplinary University Research Initiative (MURI) program administered by the Office of Naval Research under the grant number N00014-00-1-0564 and by the DoD Multidisciplinary University Research Initiative (MURI) program administered by the Air Force Office of Scientific Research under the grant number F49620-02-1-0233.

1 Introduction

From physics to biology, from weather forecasting to predicting the performance of a new processor design, researchers in many avenues of science increasingly depend on software simulations to model various realistic phenomena or hypothetical scenarios that often cannot be satisfactorily expressed analytically nor easily reproduced and observed empirically. Instead, simulation models are derived and then encoded as discrete event-driven programs: events are time-stamped messages processed in their temporal order as the simulator progresses through simulated time; event processing involves updating the simulation program state according to the given model and possibly scheduling more events in the future. In this paper, we propose a new approach for constructing discrete-event simulators that leverages a modern virtual machine to achieve performance.

1.1 Background

Due to their popularity and widespread utility, discrete event simulators have been the subject of much research (surveyed in [2, 3, 4, 5]). Systems researchers have built many types of simulation *kernels* and *libraries*, while the languages community has designed numerous *languages* specifically for simulation.

Simulation ***kernels***, including systems such as the seminal TimeWarp OS [6], transparently create a convenient simulation time abstraction. By mimicking the system call interface of a conventional operating system, one can run simulations comprised of *standard*, unmodified programs. However, since the kernel controls process scheduling, inter-process communication and the system clock, the kernel can run its applications in simulation time. For example, an application `sleep` request can be performed without delay, provided that the causal ordering of messages among communicating processes is preserved. Moreover, the kernel may transparently support concurrent execution of simulation applications and even speculative and distributed execution. The process boundary provides much flexibility.

Unfortunately, the process boundary is also a source of inefficiency [7]. Simulation ***libraries***, such as Compose [8] and others, trade away the *transparency* afforded by process-level isolation in favor of increased *efficiency*. For example, by combining the individual processes, one can eliminate the process context-switching and marshalling overheads required for event dispatch and thus increase simulation efficiency. However, various simulation functions that existed within the kernel, such as message passing and scheduling, must then be explicitly programmed in user-space. In essence, the simulation kernel and its applications are merged into a single monolithic process that contains both the simulation model as well as its own execution engine. This monolithic simulation program is more complex and littered with simulation library calls and callbacks. The library may also require certain coding practices and program structure that are not explicitly enforced by the compiler. This level of detail not only encumbers efforts to transparently parallelize or distribute the simulator, it also impedes possible high-level compiler optimizations and obscures simulation correctness. More advanced simulation kernels support a similar hybrid design through the introduction of simulation time threading, sacrificing process isolation and transparency for performance.

Finally, general-purpose simulation ***languages***, such as Simula [9], Parsec [10] and many others, are designed to simplify simulation development and to explicitly enforce the correctness of monolithic simulation programs. Simulation languages often introduce execution semantics that transparently allow for parallel and speculative execution, without any program modification. Such languages often also introduce handy constructs, such as messages and entities, that can be used to partition the application state. Constraints on simulation state and on event causality are statically enforced by the compiler, and they also permit important static and dynamic optimizations. An interesting recent example of a language-based simulation optimization is that of reducing the overhead of speculative simulation execution through the use of reverse computations [11]. However, despite these advantages, even general-purpose simulation languages are domain-specific by definition and therefore suffer from specialization relative to general-purpose computing languages, such as C, Java, Scheme, ML, etc. While they offer handy simulation-oriented features, they usually lack modern language features, such as type safety, reflection, dynamic compilation, garbage collection, and portability. They also lag in terms of general-purpose optimizations and implementation efficiency. These deficiencies only serve to perpetuate the small user-base problem, but perhaps the most significant barrier to adoption by the broader community is that existing programs need to be rewritten in order to be simulated. A similar argument (see section 3.2) applies also to high-level, domain-specific simulation languages, such as TeD for telecommunications, which enable rapid prototyping, modeling and simulator configuration for particular simulation domains.

	kernel	library	language	**JiST**
transparent	++		++	++
efficient		+	+	++
standard	++	++		++

Table 1. Trade-offs inherent to different approaches of constructing simulations.

In summary, each of these three fundamental approaches to simulation construction – kernel, library, and language – trades off a different desirable property, as shown in Table 1, where:

- **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; possibly distributing the simulation and executing it in parallel or speculatively to improve performance;
- and **transparent** implies the separation of efficiency from *correctness*; that correct simulation programs can be automatically transformed to run efficiently without the insertion of simulation-specific library calls or other program modifications. **Correctness** is an assumed pre-condition that simulation programs must compute valid and useful results, regardless of how they are constructed.

1.2 Virtual machine-based simulation

Despite a plethora of ideas and contributions to theory, languages and systems, the simulation research community has repeatedly asked itself "will the field survive?" under a perception that it had "failed to make a significant impact on the general simulation community" (see [12, 13, 10] and others). We were thus hesitant to build a new simulation system. In designing our system, we decided from the outset:

- *not* to invent a simulation language, since new languages, and especially domain-specific ones, are rarely adopted by the broader community;

- *not* to create a simulation library, since libraries often require developers to clutter their code with simulation-specific library calls and impose unnatural program structure to achieve performance; and
- *not* to develop a new system kernel or language runtime for simulation, since custom kernels or language 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, integrating the prior systems and languages approaches. 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 and its runtime.

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 surprisingly efficient. Virtual machine-based simulation provides the transparency of the kernel-based approach with the performance of a library-based approach, using language-based techniques, but within a standard language and its runtime.

It is important to note from the outset that, in this paper, we focus on maximizing *sequential* simulation performance through our new approach to simulator construction. Others, in projects such as PDNS [14], SWAN-DaSSF [15], WiPPET-TeD [16] and SWiMNet [17], have investigated algorithms and techniques to achieve scalability through distributed, concurrent, and even speculative simulation execution. These techniques can sometimes provide around an order of magnitude improvement in scale, but may require multi-processor hardware or fast inter-connects to reduce synchronization costs. More importantly, such techniques are orthogonal to the ideas of this paper. A truly scalable simulation system requires raw sequential performance as well as effective distribution and parallelism.

The paper makes two primary contributions. The first contribution is the explanation and validation of a new approach to simulator construction. We describe the basic idea in section 2 and elaborate on various extensions to the model in sections 3 and 6. Our second contribution is the design and implementation of a scalable wireless network simulator based along these principles, described in section 4. We evaluate our simulation system as well as the wireless network simulator in section 5. Section 7 contains a discussion of some design decisions and directions for future work. The related work is presented in section 8.

2 System overview and design

We begin by explaining how JiST converts a virtual machine into a simulation platform. In this section, we outline the basic system architecture and explain what it means to execute a program in *simulation time*. We then describe how JiST supports the simulation time abstraction through extensions to the basic Java object model and its execution semantics. Finally, we enumerate the more important functions of the JiST system call interface and provide a short, illustrative simulation program example.

2.1 Architecture

The JiST system consists of four distinct components: a compiler, a language runtime or virtual machine, a rewriter and a language-based simulation time kernel. Figure 1 presents the JiST architecture: a simulation is first compiled, then dynamically rewritten as it is loaded, and finally executed by the virtual machine with support from the language-based simulation time kernel.

A primary goal of JiST is to execute simulations using only a standard language and runtime. Consequently, 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 system components: the rewriter and simulation time kernel.

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 they otherwise completely preserve the existing program logic. The program transformations occur once, at load time, and do not incur rewriting overhead during execution. The rewriter also does not require

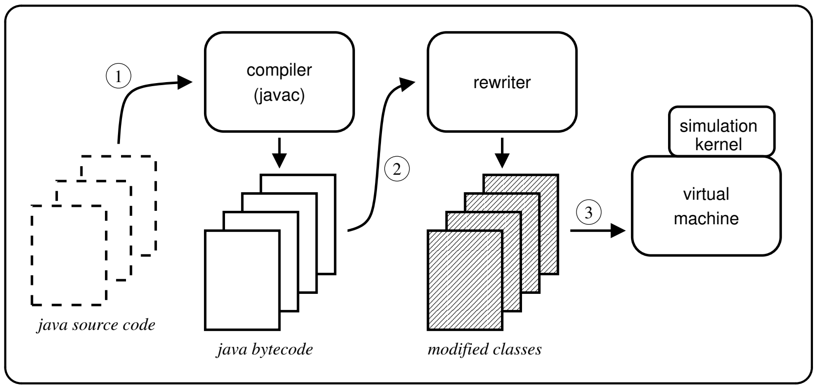


Figure 1. The JiST system architecture – simulations are (1) compiled, then (2) dynamically instrumented by the rewriter and finally (3) executed. The compiler and virtual machine are standard Java language components. Simulation time semantics are introduced by the rewriter and are supported at runtime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

source-code access, since this is a byte-code to byte-code transformation.

At runtime, the modified simulation classes interact with the JiST simulation time *kernel* through the various injected or modified operations. The JiST kernel is written entirely in Java, and it is responsible for all the runtime aspects of the simulation time abstraction. For example, it keeps track of simulation time, performs scheduling, and ensures proper synchronization.

2.2 Simulation time execution

The JiST rewriter modifies Java-based applications and runs them in *simulation time*, a deviation from the standard Java virtual machine (JVM) byte-code execution semantics [18]. Under the standard Java execution model, which we refer to as *actual time* execution, the passing of time is not explicitly linked to the progress of the application. In other words, the system clock advances regardless of how many byte-code instructions are processed. Also, the program can advance at a variable rate, since it depends not only on processor speed, but also on other unpredictable things, such as interrupts and application inputs. Moreover, the JVM does not make strong guarantees regarding timely program progress. It may decide, for example, to perform garbage collection at any point.

actual time	-	program progress and time are independent
real time	-	program progress depends on time
simulation time	-	time depends on program progress

Table 2. The relationship between program progress and time under different execution models

Under *simulation time* execution, we make the progress of time dependent on the progress of the application. The application clock, which represents simulation time, does not advance to the next discrete time point until all processing for the current simulation time has been completed. One could contrast this with *real time* execution, wherein the runtime guarantees that instructions or sets of instructions will meet given deadlines. In this case, the rate of application progress is made dependent on the passing of time. We summarize the different execution models in Table 2.

The notion of simulation time itself is not new: simulation program writers have long been accustomed to explicitly tracking the simulation time and explicitly scheduling simulation events in time-ordered queues [19]. The simulation time concept is also integral to a number of simulation languages and simulation class libraries. The novelty of the JiST system is that it embeds simulation time semantics into the standard Java language, which allows the system to transparently run the resulting simulations efficiently. Under simulation time execution, individual application byte-code instructions are processed sequentially, following the standard Java control flow semantics. However, the simulation time will remain unchanged. Application code can only advance simulation time via the `sleep(`n`)` system call. In essence, every instruction takes zero simulation time to process except for `sleep`, which advances the simulation clock forward by exactly n simulated time quanta, or ticks. In other words, the `sleep` function advances time under simulation time execution, just as it does under actual time execution. The primary difference is that, under simulation time execution, all the *other* program instructions do not have the side-effect of allowing time to pass as they are processed.

Thus, JiST is not intended to simulate the execution of arbitrary Java programs. Rather, JiST is a simulation framework that can transparently and efficiently execute simulation programs over the Java platform. JiST processes applications in their simulation-temporal order, until all queued events are exhausted or until a pre-determined ending time is reached, whichever comes first. This simulation program could be modeling anything from a wireless network to a peer-to-peer application to a new processor design. The structure of such simulation programs is described next.

2.3 Object model and execution semantics

JiST simulation programs are written in Java [20], an object-oriented language. Thus, the entire simulation program comprises numerous classes that collectively implement its logic and the state of the program is contained within objects during its execution. Interactions among object are represented syntactically as method invocations.

JiST extends this traditional programming model with the notion of simulation *entities*, defined syntactically as instances of classes that implement the empty `Entity` interface. Every simulation object must be logically contained within an entity, where we define object containment within an entity in terms of its reachability: the state of an entity is the combined state of all objects reachable from it. Thus, although entities are regular objects within the virtual machine at runtime, they serve to logically encapsulate application objects, as shown in Figure 2(a). Entities are components of a simulation and represent the granularity at which the JiST kernel manages a running simulation.

Each entity has its own simulation time and may progress through simulation time independently. Thus, an entity cannot share its state with any other entity, otherwise there could be an inconsistency in the state of the simulation. In other words, each (mutable) object of the simulation must be contained within exactly one entity. Since Java is a safe language, this constraint is sufficient to partition the simulation into a set of non-overlapping entities and also prevents unmediated communication across entity boundaries.

All instructions and operations *within* an entity follow the regular Java control flow and semantics. They are entirely opaque to the JiST infrastructure. Specifically, object method invocations remain unchanged. The vast majority of the entity code is involved with encoding the logic of the simulation model and is entirely unrelated 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 JiST system calls.

In contrast, invocations on entities correspond to simulation events. The execution semantics are that method invocations on entities are non-blocking. They are merely queued at their point of invocation. The invocation is actually performed on the callee (or target) entity only when it

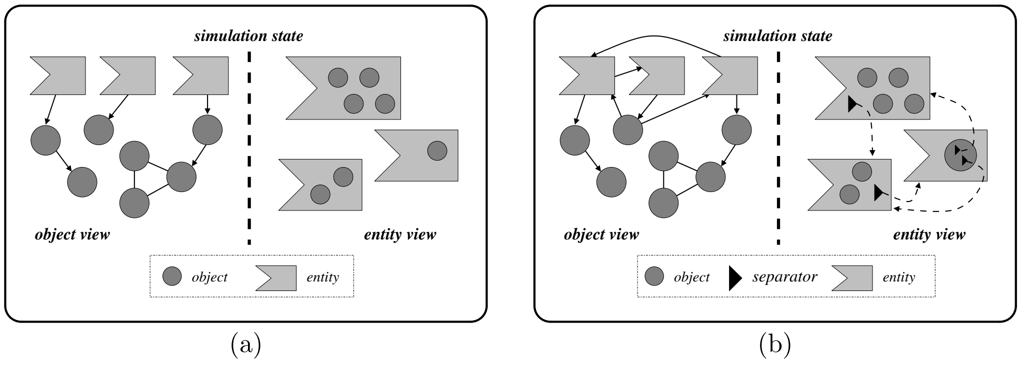


Figure 2. (a) Simulation programs are partitioned into entities along object boundaries. Thus, entities do not share any application state and can independently progress through simulation time between interactions. (b) At runtime, entity references are transparently replaced with separators, which both preserves the separation of entity state and serves as a convenient point to insert functionality.

reaches the same simulation time as the calling (or source) entity. In other words, cross-entity method invocations act as synchronization points in simulation time. Or, from a language-oriented perspective, an entity method is like a coroutine, albeit scheduled in simulation time. This is a convenient abstraction in that it eliminates the need for an explicit simulation event queue. It is the JiST kernel that actually runs the event loop, which processes the simulation events. The kernel invokes the appropriate method for each event dequeued in its simulation time order and executes the event to completion without continuation.

However, in order to invoke a method on another entity – to send it an event – the calling entity must hold some kind of reference to the target entity, as depicted in Figure 2(b). We, therefore, distinguish between object references and entity references. All references to a given (mutable) object must originate from within the same entity. References to entities are free to originate from any entity, including from objects within any entity. The rationale is that object references imply inclusion within the state of an entity, whereas entity references represent channels along which simulation events are transmitted. As a consequence, entities do not nest, just as regular Java objects do not.

We reintroduce the separation of entities at runtime by transparently replacing all entity references within the simulation byte-code with special objects, called *separators*. The separator object identifies a particular target entity, but without referencing it directly. Rather, separators store a unique entity identifier that is generated by the kernel for each simulation entity during its initialization. Separators can be held in local variables, stored in fields or objects or passed as parameters to methods, just like the regular object references that they replace. Since replacement occurs across the entire simulation byte-code, it remains type-safe.

Due to this imposed separation, we guarantee that interactions among entities can only occur via the JiST kernel. This is similar in spirit to the JKernel design [21] in that it provides language-based protection and zero-copy inter-entity communication. However, JKernel is designed to provide traditional system services, such as process-level protection, within a safe-language environment, whereas JiST is designed explicitly for simulation. For example, whereas JKernel utilizes native Java threads for concurrency, JiST introduces entities. Entities provide thread-free event-based simulation time concurrency, which facilitates scalable simulation.

The separators, in effect, represent an application state-time boundary around each entity, similar to a TimeWarp [6] process, but at a finer granularity. Separators are a convenient point to insert additional simulation functionality. For example, by tracking the simulation time of each individual entity, these separators allow for concurrent execution. Or, by adding the ability to checkpoint entities, the system can support speculative execution as well. Finally, separators also provide a convenient point for the distribution of entities across multiple machines. In a distributed simulation, the separators function as remote stubs and transparently maintain a convenient abstraction of a single system image. Separators can transparently store and track the location of entities as they migrate among machines in response to fluctuating processor, memory, and network loads.

The role of the simulation developer, then, is to write the simulation model in regular Java and to partition the program into multiple entities along reasonable application boundaries. This is akin to partitioning the application into separate classes. The JiST infrastructure will transparently execute the program efficiently, while retaining the simulation time semantics.

This model of execution, known as the *concurrent object model*, is similar to, for example, the Compose [8] simulation library. It invokes a method for every message received and executes it to completion. This is in contrast to the *process model* that is used, for example, in the Parsec language [10], wherein explicit send and receive operations are interspersed in the code. In the process model, each entity must store a program-counter and a stack as part of its state.

JistAPI.java

```java
package jist.runtime;
class JistAPI {
   interface Entity { }
   long getTime();
   void sleep(long ticks);
   void end();
   void endAt(long time);
   void run(int type, String name, String[] args, Object props);
   void runAt(Runnable r, long time);
   void setSimUnits(long ticks, String name);
   interface Timeless { }
   interface Proxiable { }
   Object proxy(Object proxyTarget, Class proxyInterface);
   class Continuation extends Error { }
   Channel createChannel();
   interface CustomRewriter { JavaClass process(JavaClass jcl); }
   void installRewrite(CustomRewriter rewrite);
   interface Logger { void log(String s); }
   void setLog(JistAPI.Logger logger);
   void log(String s);
   Entity THIS;
   EntityRef ref(Entity e);
}
```

Figure 3. The partial JiST system call interface shown above is exposed at the language level via the `JistAPI` class. The rewriter replaces these with their runtime implementations.

Unlike Compose, message sending in JiST is embedded in the language and does not require a simulation library. Unlike Parsec, JiST embeds itself within the Java language and does not require new language constructs. With the introduction of continuations in section 6.2, we will even allow these two simulation models to co-exist.

2.4 Simulation kernel interface

JiST simulations run atop the simulation kernel and interact with it via a small API. The entire API is exposed at the language level via the `JistAPI` class listed partially in Figure 3 and explained below. The remainder of the API will be explained as the corresponding concepts are introduced.

The `Entity` interface tags a simulation object as an entity, which means that invocations on this object follow simulation time semantics: method invocations become events that are queued for delivery at the simulation time of the caller. The `getTime` call returns the current hello.java

```
 1 import jist.runtime.JistAPI;
 2 class hello implements JistAPI.Entity {
 3   public static void main(String[] args) {
 4     System.out.print("start simulation");
 5     hello h = new hello();
 6     h.myEvent();
 7   }
 8   public void myEvent() {
 9     JistAPI.sleep(1);
10     myEvent();
11     System.out.print(" hello world, t="+JistAPI.getTime());
12   }
13 }
```

Figure 4. The simplest of simulations consists of one entity that emits a message at each time step.

simulation time of the calling entity, which is the time of the current event being processed plus any additional sleep time. The `sleep` call advances the simulation time of the calling entity. The `endAt` call specifies when the simulation should end. The `THIS` self-referencing entity reference is analogous to the Java `this` object self-reference. It refers to the entity for which an event is currently being processed and is rarely needed. The `ref` call returns a separator stub of a given entity. All statically detectable entity references are automatically converted into separator stubs by the rewriter. It is included only to deal with rare instances when entity types might be created dynamically and for completeness.

These basic simulation primitives allow us to write simulators. 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 4.[*] This simplest of simulations highlights some important points. To begin, the `hello` class is an entity, since it implements the `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 JiST rewriting) then the program would

[*]Many other, longer examples may be found online [1].

abort with 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 with the entity simulation time (line 11). Instead of a stack overflow, the program runs in constant stack space and the output is:

```
simulation start  hello world, t=1  hello world, t=2  hello world, t=3  etc.
```

3 Optimizations

Having introduced the fundamental simulation time transformation, we now discuss two performance-related extensions to the model: timeless objects and reflection-based configuration. These additions are orthogonal to the simulation program. We annotate the underlying simulation code, if at all necessary, and then perform high-level optimizations and cross-cutting transformations with the rewriter, akin to aspect-oriented programming [22]. Other such extensions to the model, which are designed specifically to simplify simulation development, are deferred to section 6.

3.1 Zero-copy semantics

Our first extension is the notion of *timeless* objects. A timeless object is defined as one that will not change over time. Knowing that a value is temporally stable allows the system to safely pass it across entities by reference, rather than by copy. The system may be able to statically infer that an object is transitively open-world immutable [23] and automatically add the timeless labels. However, any static analysis will be overly conservative at times. Thus, one can explicitly request zero-copy semantics by using the `Timeless` interface to tag an object, which implies that the tagged object will not be modified at any time after references to it escape an entity boundary. Thus, the addition of a single tag, or the automatic detection of the timeless property, affects all the events within the simulation that contain objects of this type. The explicit tagging facility should be exercised with care, since individual entities may progress at different rates through simulation time, and this can result in a temporal inconsistency within the simulation state.

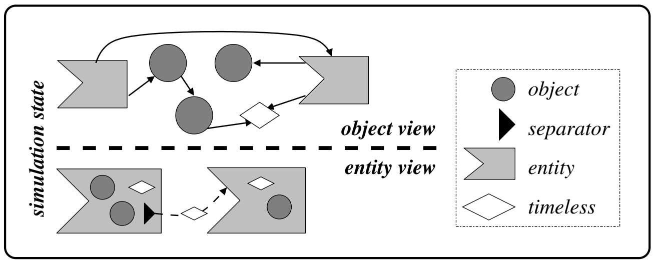


Figure 5. Objects passed across entities should be timeless – in other words, should hold temporally stable values – to prevent temporal inconsistencies in the simulation state. Sharing timeless objects among entities is an effective way to conserve simulation memory and using zero-copy semantics improves simulation throughput.

The timeless tag is also useful for sharing state among entities to reduce simulation memory consumption, as depicted in Figure 5. For example, network packets are defined to be timeless in SWANS, our wireless network simulator, in order to prevent unnecessary duplication: broadcasted network packets are therefore not copied for every recipient, nor are they copied in the various sender retransmit buffers. Similarly, one can safely share object replicas across different instances of a simulated peer-to-peer application.

3.2 Reflection-based configuration

An important consideration in the design of simulators is configurability: the desire to reuse the simulator for many different experiments. However, this can adversely affect performance. Configuration is usually supported either at the *source-code* level, via *configuration files*, or with *scripting* languages.

Source-level configuration entails the recompilation of the simulation program before each run with hard-coded simulation parameters and linkage with a small driver program for simulation initialization. This approach to configuration is flexible and runs efficiently, because the compiler can perform constant propagation and other important optimizations on the generic simulation code to produce a specialized and efficient executable. However, it requires recompilation on each run. The use of ***configuration files*** eliminates the need for recompilation. The configuration is read and parsed by a generic driver program as it initializes the simulation. This option is not only brittle and limited to pre-defined configuration options,

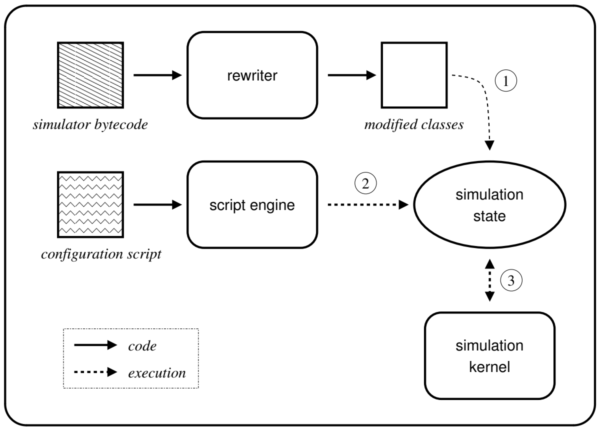


Figure 6. JiST can easily provide multiple scripting interfaces to configure its simulations without source modification, memory overhead, or loss of performance. (1) As before, the simulation classes are loaded and rewritten on demand. (2) The script engine configures the simulation using reflection and may even dynamically compile the script byte-code for performance. (3) The simulation then runs as before, interacting with the kernel as necessary.

it eliminates opportunities for static compiler optimizations. The ***script-based configuration*** approach is championed by ns2 [19], a popular network simulator. A scripting language interpreter – Tcl, in the case of ns2 – is backed by the compiled simulation runtime, so that script variables are linked to simulation values, and a script can then be used to instantiate and initialize the various pre-defined simulation components. Unfortunately, the linkage between the compiled simulation components and the configuration scripts can be difficult to establish. In ns2, it is achieved manually via a programming pattern called *split objects*, which requires a language binding that channels information in objects within the compiled space to and from objects in the interpreted space. This not only clutters the core simulation code, but it is also inefficient, because it duplicates information. Furthermore, script performance depends heavily on this binding. The choice of combining C with Tcl in ns2, for example, imposes excessive string manipulation and leads to long configuration times. More importantly, this approach eliminates static optimization opportunities, which hurts performance. It also results in the loss of both static and dynamic type information across the compiled-interpreted interface, thereby increasing the potential for error.

In contrast, JiST-based simulations enjoy both the flexibility of script-based configuration and the performance advantages of source-level configuration. The scripting functionality comes “for free” in JiST. It does not require any additional code in the simulation components, nor any additional memory. And, the script can configure a simulation just as quickly as a custom driver program. This is because Java is a dynamic language that supports reflection. As illustrated in Figure 6, the access that the script engine has to the simulation state is just as efficient and expressive as the compiled driver program. A script engine can query and update simulation values by reflection for purposes of tracing, logging, and debugging, and it can also dynamically pre-compile the driver script directly to byte-code for efficient execution. The simulation byte-code itself is compiled and optimized dynamically, as the simulation executes. Thus, simulation configuration values are available to the Java optimizer and allow it to generate more efficient and specialized code. The information available to the optimizer at runtime is a super-set of what is available to a static simulation compiler. Finally, while the scripting language environment may support a more relaxed type system, the type-safety of the underlying simulation components is still guaranteed by the virtual machine, facilitating earlier detection of scripting errors.

The script functionality is exposed via the `JistAPI`, so that simulators may also embed domain-specific configuration languages. JiST supports the BeanShell engine, with its Java syntax, and also the Jython engine, which interprets Python scripts. Java-based engines for other languages, including Smalltalk, Tcl, Ruby, Scheme and JavaScript, also exist and can be integrated. Likewise, high-level, domain-specific simulation languages, such as TeD, and also graphical tools can be used to simplify the task of modeling and simulator configuration. Thus, specialization is possible without loss of performance.

4 Case study: SWANS - Wireless network simulation

Wireless networking research is fundamentally dependent upon simulation. Analytically quantifying the performance and complex behavior of even simple protocols on a large scale is often imprecise. On the other hand, performing actual experiments is onerous: acquiring hundreds of devices, managing their software and configuration, controlling a distributed experiment and aggregating the data, possibly moving the devices around, finding the physical space for such an experiment, isolating it from interference and generally ensuring *ceteris*

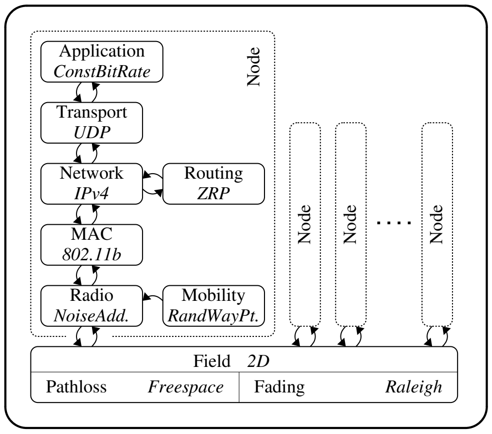


Figure 7. The SWANS simulator consists of event-driven components that can be configured and composed to form a meaningful wireless network simulation. Different classes of components are shown in a typical arrangement together with specific instances of component implementations in italics.

paribus, are but some of the difficulties that make empirical endeavors daunting. Consequently, the majority of publications in this area are based entirely upon simulation. Other related and applicable areas of recent research interest include overlay networks [24] and peer-to-peer applications [25, 26, 27, 28].

At a minimum, we would like to simulate networks of many thousands of nodes. However, even though a few parallel discrete event simulation environments have been shown to scale to networks of beyond 10^4 nodes, slow sequential network simulators remain the norm [29]. In particular, most published ad hoc network results are based on simulations of few nodes only (usually fewer than 500 nodes), for a short duration, and over a small geographical area. Larger simulations usually compromise on simulation detail. For example, they may simulate only at the packet level without considering the effects of signal interference. Others reduce the complexity of the simulation by curtailing the simulation duration, reducing the node density, or restricting mobility. Our goal in this work is to improve the state of the art in this regard.

Therefore, as a proof of the JiST approach as well as a research tool, we constructed SWANS, a **S**calable **W**ireless **A**d hoc **N**etwork **S**imulator, atop the JiST platform. The SWANS software is organized as independent software components that can be composed to form complete wireless simulations, as shown in Figure 7. Its capabilities are similar to ns2 [19] and GloMoSim [30], two popular wireless network simulators. There are components that implement different types of applications; networking, routing and media access protocols; radio transmission, reception and noise models; signal propagation and fading models; and node mobility models. Instances of each component type are shown italicized in the figure. Nothing within the design of JiST is specific to the simulation of wireless networks; SWANS is merely an exemplary application.

Notably, the development of SWANS has been relatively painless. Since JiST inter-entity message creation and delivery is implicit, as well as message garbage collection and typing, the code is compact and intuitive. Components in JiST consume less than half of the code (in uncommented line counts) of comparable components in GloMoSim, which are already smaller than their counterpart implementations in ns2.

4.1 Design highlights

Every SWANS component is encapsulated as a JiST entity: it stores it own local state and interacts with other components via exposed event-based interfaces. SWANS contains components for constructing a node stack, as well components for a variety of mobility models and field configurations. This pattern simplifies simulation development by reducing the problem to creating relatively small, event-driven components. It also explicitly partitions the simulation state and the degree of inter-dependence between components, unlike the design of ns2 and GloMoSim. It also allows components to be readily interchanged with suitable alternate implementations of the common interfaces and for each simulated node to be independently configured. Finally, it also confines the simulation communication pattern. For example, `Application` or `Routing` components of different nodes cannot communicate directly. They can only pass messages along their own node stacks.

Consequently, the elements of the simulated node stack above the `Radio` layer become trivially parallelizable, and may be distributed with low synchronization cost. In contrast, different `Radio`s do contend (in simulation time) over the shared `Field` entity and raise the synchronization cost of a concurrent simulation execution. To reduce this contention in a distributed simulation, the simulated field may be partitioned into non-overlapping, cooperating `Field` entities along a grid.

It is important to note that, in JiST, communication among entities is very efficient. The design incurs no serialization, copy, or context-switching cost among co-located entities, since the Java objects contained within events are passed along by reference via the simulation time kernel. Simulated network packets are actually a chain of nested objects that mimic the chain of packet headers added by the network stack. Moreover, since the packets are timeless by design, a single broadcasted packet can be safely shared among all the receiving nodes and the very same object sent by an `Application` entity on one node will be received at the `Application` entity of another node. Similarly, if we use TCP in our node stack, then the same object will be referenced in the sending node's TCP retransmit buffer. This design conserves memory, which in turn allows for the simulation of larger network models.

Dynamically created objects such as packets can traverse many different control paths within the simulator and can have highly variable lifetimes. The accounting for when to free unused packets is handled entirely by the garbage collector. This not only simplifies the memory management protocol, but also eliminates a common source of memory leaks that can accumulate over long simulation runs.

The partitioning of node functionality into individual, fine-grained entities provides an additional degree of flexibility for distributed simulations. The entities can be *vertically* aggregated, as in GloMoSim, which allows communication along a network stack *within* a node to occur more efficiently. However, the entities can also be *horizontally* aggregated to allow communication *across* nodes to occur more efficiently. In JiST, this reconfiguration can happen without any change to the entities themselves. The distribution of entities across physical hosts running the simulation can be changed dynamically in response to simulation communication patterns and it does not need to be homogeneous.

4.2 Embedding Java-based network applications

SWANS has a unique and important advantage over existing network simulators. It can run regular, unmodified Java network applications over the simulated network, thus allowing for the inclusion of existing Java-based software, such as web servers, peer-to-peer applications and application-level multicast protocols. These applications do not merely send packets to the simulator from other processes. They operate in simulation time within the same JiST process space, allowing far greater scalability. As another example, one could perform a similar transformation on Java-based database engines or file-system applications to generate accesses within an disk simulator.

We achieve this integration via a special `AppJava` application entity designed to be a harness for Java applications. This harness inserts an additional rewriting phase into the JiST kernel, which substitutes SWANS socket implementations for any Java counterparts that occur within the application. These SWANS sockets have identical semantics, but send packets through the simulated network. Specifically, the input and output methods are still blocking operations (see section 6.2). To support these blocking semantics, JiST automatically modifies the necessary application code into continuation-passing style, which allows the application to operate within the event-oriented simulation time environment.

4.3 Efficient signal propagation using hierarchical binning

Modeling signal propagation within the wireless network is strictly an application-level issue, unrelated to JiST performance. However, doing so efficiently is essential for scalable wireless simulation. When a simulated radio entity transmits a signal, the SWANS `Field` entity must deliver that signal to all radios that could be affected, after considering fading, gain, and pathloss. Some small subset of the radios on the field will be within reception range and a few more radios will be affected by the interference above some sensitivity threshold. The remaining majority of the radios will not be tangibly affected by the transmission.

ns2 and GloMoSim implement a naïve signal propagation algorithm, which uses a slow, $O(n)$, linear search through *all* the radios to determine the node set within the reception neighborhood of the transmitter. This clearly does not scale as the number of radios increases. ns2 has recently been improved with a grid-based algorithm [31]. We have implemented both of these in SWANS. In addition, we have a new, more efficient algorithm that uses *hierarchical* binning. The spatial partitioning imposed by each of these data structures is depicted in Figure 8.

In the grid-based or flat binning approach, the field is sub-divided into a grid of node bins. A node location update requires constant time, since the bins divide the field in a regular manner. The neighborhood search is then performed by scanning all bins within a given distance from the signal source. While this operation is also of constant time, given a sufficiently fine grid, the constant is sensitive to the chosen bin size: bin sizes that are too large will capture too many nodes and thus not serve their search-pruning purpose; bin sizes that are too small will require the scanning of many empty bins, especially at lower node densities. A reasonable bin

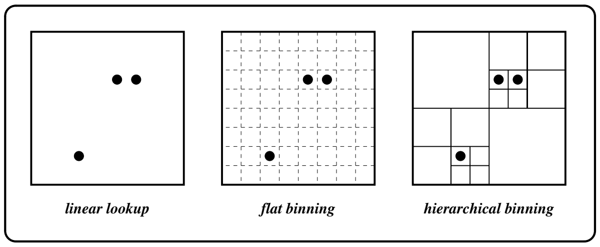


Figure 8. Efficient signal propagation is critical for wireless network simulation performance. Hierarchical binning of radios allows location updates to be performed in expected amortized constant time and the set of receiving radios to be computed in time proportional to its size.

size is one that captures a small number of nodes per bin. Thus, the bin size is a function of the local radio density and the signal propagation radius. However, these parameters may change in different parts of the field, from radio to radio, and even as a function of time, for example, as in the case of power-controlled transmissions.

We improve on the flat binning approach. Instead of a flat sub-division, the hierarchical binning implementation recursively divides the field along both the x and y-axes. The node bins are the leaves of this balanced, spatial decomposition tree, which is of height equal to the number of divisions, or $log_4(\frac{field\ size}{bin\ size})$. The structure is similar to a quad-tree, except that the division points are not the nodes themselves, but rather fixed coordinates. Note that the height of the tree changes only logarithmically with changes in the bin or field size. Furthermore, since nodes move only a short distance between updates, the expected amortized height of the common parent of the two affected node bins is $O(1)$. This, of course, is under the assumption of a reasonable node mobility model that keeps the nodes uniformly distributed. Thus, the amortized cost of updating a node location is constant, including the maintenance of inner node counts. When scanning for node neighbors, empty bins can be pruned as we descend spatially. Thus, the set of receiving radios can be computed in time proportional to the number of receiving radios. Since, at a minimum, we will need to simulate delivery of the signal at each simulated radio, the algorithm is asymptotically as efficient as scanning a cached result, as proposed in [17], even assuming perfect caching. But, the memory overhead of hierarchical binning is minimal. Asymptotically, it amounts to $\lim_{n\rightarrow\infty}\sum_{i=1}^{log_4 n}\frac{n}{4^i} = \frac{n}{3}$.

The memory overhead for function caching is also $O(n)$, but with a much larger constant. Furthermore, unlike the cases of flat binning or function caching, the memory accesses for hierarchical binning are tree structured and thus exhibit better locality.

5 Evaluation

Having explained the fundamental elements of the JiST and SWANS designs, we now turn to performance evaluation. Conventional wisdom regarding language performance [32] argues against implementing our system in Java. In fact, the vast majority of existing simulation systems have been written in C and C++, or their derivatives. Nevertheless, we show in this section that JiST and SWANS perform surprisingly well: aggressive profile-driven optimizations combined with the latest Java runtimes result in a high-performance simulation system. We compare SWANS with the two most popular ad hoc network simulators: ns2 and GloMoSim. We selected these because they are widely used, freely available sequential network simulators designed in the systems-based and language-based approaches, respectively.

The ns2 network simulator [19] has a long history with the networking community, is widely trusted, and has been extended to support mobility and wireless networking protocols. It is designed as a monolithic, sequential simulator, constructed using the simulation library approach. ns2 uses a clever “split object” design, which allows Tcl-based script configuration of C-based object implementations. Researchers have also extended ns2 to conservatively parallelize its event loop [14]. However, this technique has proved primarily beneficial for distributing ns2’s considerable memory requirements. Based on numerous published results, it is not easy to scale ns2 beyond a few hundred simulated nodes. Simulation researchers have shown ns2 to scale, with difficulty and substantial hardware resources, to simulations of a few thousand nodes [29].

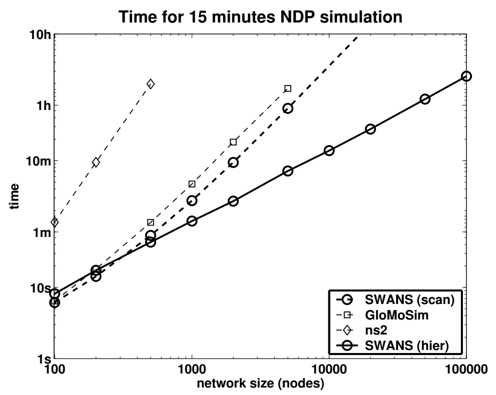
GloMoSim [30] is a newer simulator written in Parsec [10], a highly-optimized C-like simulation language. GloMoSim has recently gained popularity within the wireless ad hoc networking community, since it was designed specifically for scalable network simulation. For example, to overcome Parsec’s large per-entity memory requirements, GloMoSim implements a technique called “node aggregation” to conserve memory, by combining the state of multiple simulated network nodes into a single Parsec entity. Parallel GloMoSim has been shown to scale to 10,000 nodes on multi-processor machines. The sequential version of GloMoSim is freely available and the conservatively parallel version is commercialized as QualNet [33].

In this section, we present macro-benchmark results running full SWANS simulations. We then show micro-benchmark results that highlight the throughput and memory advantages of JiST. Unless otherwise noted, the following measurements were taken on a 2.0 GHz Intel Pentium 4 single-processor machine with 512 MB of RAM and 512 KB of L2 cache, running the version 2.4.20 stock Redhat 9 Linux kernel with glibc v2.3. We used the publicly available versions of Java 2 JDK (v1.4.2), Parsec (v1.1.1), GloMoSim (v2.03) and ns2 (v2.26). Each data point presented represents an average of at least five runs for the shorter time measurements. All tests were also performed on a second machine – a more powerful and memory rich dual-processor – giving identical absolute memory results and relative results for throughput (i.e. scaled with respect to processor speed).

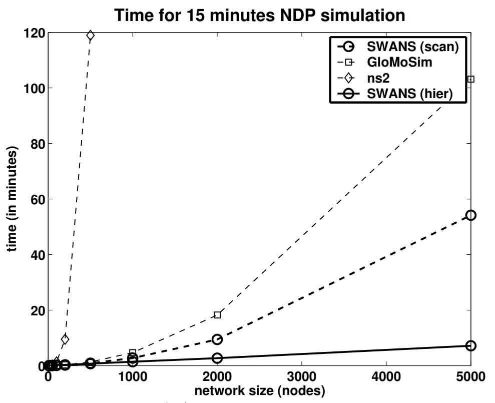
5.1 Macro-benchmarks

In the following experiment, we benchmarked JiST running a full SWANS ad hoc wireless network simulation. We measured the performance of simulating an ad hoc network of nodes running a UDP-based beaconing node discovery protocol (NDP) application. Node discovery protocols are an integral component of many ad hoc network protocols and applications [34, 35]. Also, this experiment is representative both in terms of code coverage and network traffic: it utilizes the entire network stack and transmits over every link in the network every few seconds. However, the experiment is still simple enough that we have high confidence of simulating *exactly* the same operations across the different platforms – SWANS, GloMoSim and ns2, – which permits comparison and is difficult to achieve with more complex protocols. Finally, we were also able to validate the simulation results against analytical estimates.

We constructed the following identical scenario in each of the simulation platforms. The application at each node maintains a local neighbor table and beacons every 2 to 5 seconds, chosen from a uniform random distribution. Each wireless node is placed randomly in the network coverage area and moves with random-waypoint mobility [35] at speeds of 2 to 10 meters per second selected at random and with pause times of 30 seconds. Mobility in ns2 was turned off, because the pre-computed trajectories resulted in excessively long configuration times and memory consumption. Each node is equipped with a standard radio configured with typical 802.11b signal strength parameters. The simulator accounts for free-space path loss with ground reflection and Rayleigh fading. We ran simulations with widely varying numbers

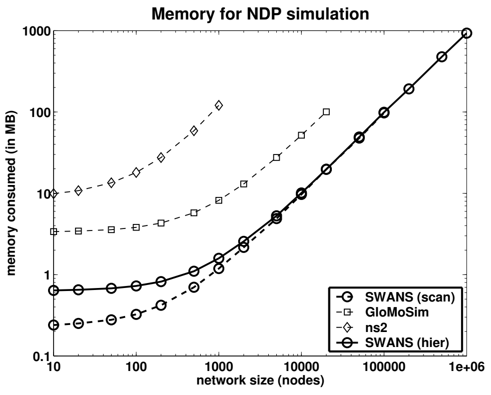


(a) log-log scale



(b) linear scale

Figure 9. SWANS significantly outperforms both ns2 and GloMoSim in simulations of the node discovery protocol.



nodes	simulator	time	memory
500	**SWANS**	**54 s**	**700 KB**
	GloMoSim	82 s	5759 KB
	ns2	7136 s	58761 KB
	SWANS-hier	*43 s*	*1101 KB*
5,000	**SWANS**	**3250 s**	**4887 KB**
	GloMoSim	6191 s	27570 KB
	SWANS-hier	*430 s*	*5284 KB*
50,000	**SWANS**	**312019 s**	**47717 KB**
	SWANS-hier	*4377 s*	*49262 KB*

Figure 10. SWANS can simulate larger network models due to its more efficient use of memory.

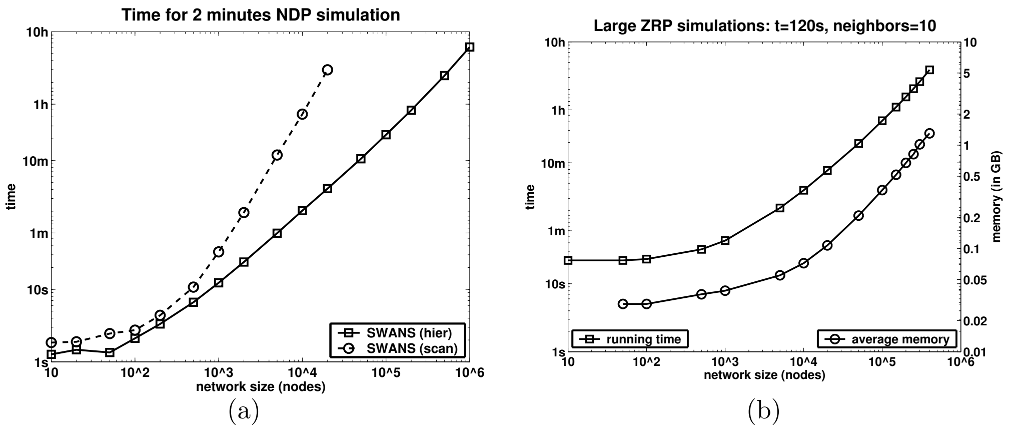


Figure 11. (a) SWANS scales to networks of 10^6 wireless nodes. The figure shows the time for a sequential simulation of a node discovery protocol in a wireless ad hoc network running on a commodity machine. (b) Time and space consumed for a simulation of a more complex network protocol, the Zone Routing Protocol.

of nodes, keeping the node density constant, such that each node transmission is received, on average, by 4 to 5 nodes and interferes with approximately 35 others. Above each radio, we constructed a stack of 802.11b MAC, IPv4 network, UDP transport, and NDP application entities.

We ran this network model for 15 simulated minutes and measured overall memory and time required for the simulation. For memory, we included the base process memory, the memory overhead for simulation entities, and all the simulation data at the beginning of the simulation. For time, we included the simulation setup time, the event processing overheads, and the application processing time.

The throughput results are plotted both on log-log and linear scales in Figure 9. As expected, the simulation times are quadratic functions of n, the number of nodes, when using the naïve signal propagation algorithm. Even without node mobility, ns2 is highly inefficient. SWANS outperforms GloMoSim by a factor of 2. SWANS-hier uses the improved hierarchical binning algorithm to perform signal propagation instead of scanning through all the radios. As expected, SWANS-hier scales linearly with the number of nodes.

The memory footprint results are plotted in Figure 10 on log-log scale. JiST is more efficient than GloMoSim and ns2 by almost an order and two orders of magnitude, respectively. This allows SWANS to simulate much larger networks. The memory overhead of hierarchical binning is asymptotically negligible. As a point of reference, regularly published results of a few hundred wireless nodes occupy more than 100 MB, and simulation researchers have scaled ns2 to around 1,500 non-wireless nodes using a 1 GB process [14, 36].

Next, we tested SWANS with some very large networks. We ran the same simulations on dual-processor 2.2GHz Intel Xeon machines (though only one processor was used) with 2GB of RAM running Windows 2003. The results are plotted in Figure 11(a) on a log-log scale. We show SWANS both with the naïve propagation algorithm and with hierarchical binning, and we observe linear behavior for the latter in all simulations up to networks of one million nodes. The 10^6 node simulation consumed just less than 1GB of memory on initial configuration, ran with an average footprint of 1.2GB (fluctuating due to delayed garbage collection), and completed within $5\frac{1}{2}$ hours. This exceeds previous ns2 and GloMoSim results by two orders of magnitude, using only commodity hardware.

Finally, we also provide simulation performance metrics for a simulation of the Zone Routing Protocol (ZRP), a more complex and realistic wireless network protocol. ZRP requires some routing table storage at each node, which increases the per node memory requirements, and also performs considerably more network communication to determine routes. Figure 11(b) shows that we were able to simulation half a million node within five hours, and shows linear increases in both time and space relative to the network size.

5.2 Event throughput

Having presented macro-benchmark results, we now evaluate JiST event processing throughput and memory consumption for both simulation entities and events. High event throughput is essential for scalable discrete event simulation. Thus, in the following micro-benchmark, we measured the performance of each of the simulation engines in performing a tight simulation event loop. We began the simulations at time zero, with an event scheduled to do nothing but schedule another identical event in the subsequent simulation time step. We ran each simulation for n simulation time quanta, over a wide range of n and measured the actual time elapsed. Note that, in performing these no-op events, we eliminate any variation due to application-specific processing and are able to observe just the *overhead* of the underlying

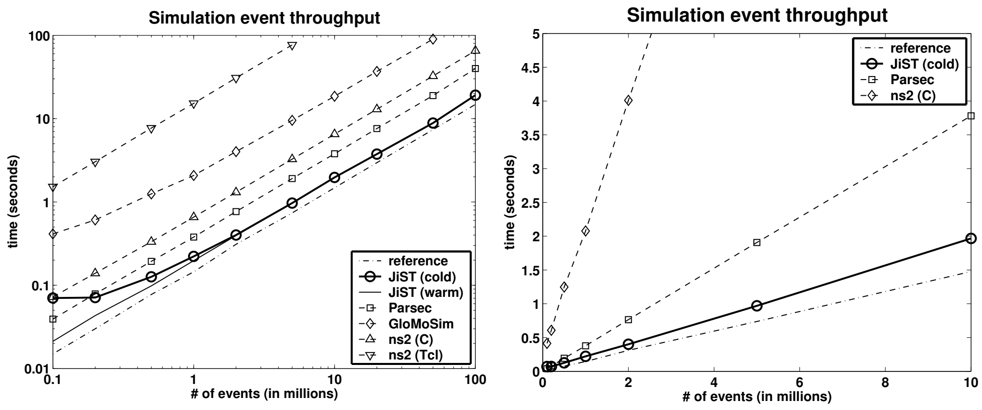


Figure 12. JiST has higher event throughput and comes within 30% of the reference lower bound program. The kink in the JiST curve in the first fraction of a second of simulation is evidence of JIT compilation and optimization at work.

event processing.

Equivalent and efficient benchmark programs were written in each of the systems. We briefly describe how each of these equivalent programs was implemented within each system. The JiST program looks much like the "hello world" program presented earlier. The Parsec program sends null messages among native Parsec entities using the special `send` and `receive` statements. The GloMoSim program considers the overhead of the node aggregation mechanism built over Parsec. It is implemented as an application component that sends messages to itself. Both the Parsec and GloMoSim tests are compiled using `pcc -O3`, the most optimized Parsec compiler setting. ns2 utilizes a split object model, allowing method invocations from either C or Tcl. The majority of the performance critical code, such as packet handling, is written in C, leaving mostly configuration operations for Tcl. However, there remain some important components, such as node mobility, that depend on Tcl along the critical path. Consequently, we ran two tests: the ns2-C and ns2-Tcl tests correspond to events scheduled from either of the languages. ns2 simulation performance lies somewhere between these two widely divergent values, dependent on how frequently each language is employed during a given simulation. Finally, we developed a reference test to obtain a lower bound on the computation. It is a program, written in C and compiled with `gcc -O3`, that merely inserts

5×10^6 events	time (sec)	vs. reference	vs. JiST
reference	0.738	1.00x	0.76x
JiST	**0.970**	**1.31x**	**1.00x**
Parsec	1.907	2.59x	1.97x
ns2-C	3.260	4.42x	3.36x
GloMoSim	9.539	12.93x	9.84x
ns2-Tcl	76.558	103.81x	78.97x

(a)

memory	entity	event
JiST	**36 B**	**36 B**
GloMoSim	36 B	64 B
ns2	544 B	40 B*
Parsec	28536 B	64 B

(b)

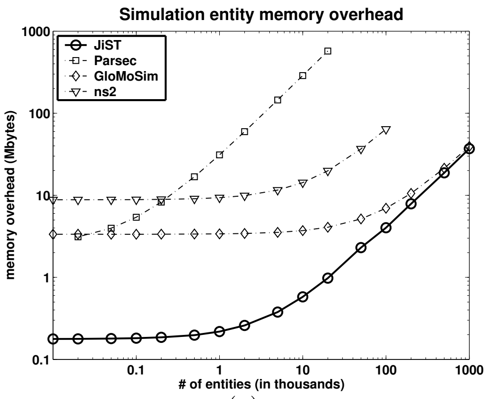
Table 3. (a) Time to perform 5 million events, normalized against both the baseline and JiST. (b) Per entity and per event memory *overhead* – i.e., *without* including memory for any simulation data.

and removes elements from an efficient implementation of an array-based priority queue.

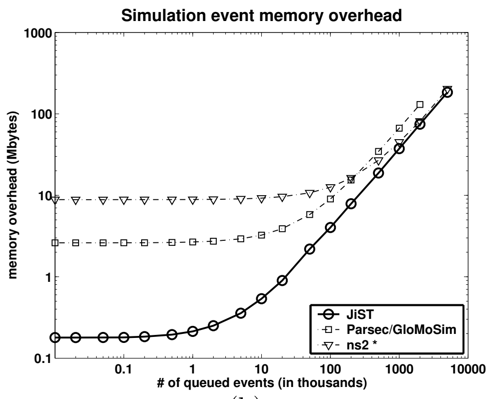
The results are plotted in Figure 12, as log-log and linear scale plots. As expected, all the simulations run in time linear with respect to the number of events, n. A counter-intuitive result is that JiST out-performs all the other systems, including the compiled ones. It also comes within 30% of the reference measure of the computational lower bound, even though it is written in Java. This achievement is due to the impressive JIT dynamic compilation and optimization capabilities of the modern Java runtime. The effect of the dynamic optimizations can actually be seen as a kink in the JiST curve during the first fraction of a second of simulation. To confirm this, we warmed the JiST test with 10^6 events (or, for two tenths of a second) and observed that the kink disappears. The linear-scale plot shows that the time spent on dynamic optimizations is negligible. Table 3(a) shows the time taken to perform 5 million events in each of the measured simulation systems and also those figures normalized against both the reference program and JiST performance. JiST is twice as fast as both Parsec and ns2-C. GloMoSim and ns2-Tcl are one and two orders of magnitude slower, respectively.

5.3 Memory utilization

Another important resource that may limit scalability is memory. In many simulations, memory is the critical scalability-limiting factor, since it establishes an upper bound on the size of the simulated model. We thus measured the memory consumed by simulation entities and by queued simulation events in each of the systems. Measuring the memory usage of entities involves the allocation of n empty entities and observing the size of the operating system process over a wide range of n. Similarly, we queue a large number of events and observe



(a)



(b)

Figure 13. JiST allocates entities efficiently: comparable to GloMoSim at 36 bytes per entity, and over an order of magnitude less that Parsec or ns2. JiST allocates events efficiently: comparable to ns2 (in C) at 36 bytes per queued event and half the size of events in Parsec and GloMoSim. (*) Events scheduled in ns2 via Tcl will allocate a split object and thereby incur the same memory overhead.

their memory requirements. In the case of Java, we invoke a garbage collection sweep before requesting an internal memory count. Note also that this benchmark, as before, measures the memory *overhead* imposed by the simulation system. The entities and events are empty: they do not carry any application data.

The entity and event memory results are plotted on log-log scales in Figure 13. The base memory footprint of each of the systems is less than 10 MB. Asymptotically, the process footprint increases linearly with the number of entities or events, as expected. (a) – JiST performs well with respect to memory requirements for simulation entities. It performs comparably with GloMoSim, which uses node aggregation specifically to reduce Parsec's memory consumption. A GloMoSim "entity" is merely a heap-allocated object containing an aggregation identifier and an event-scheduling priority queue. In contrast, each Parsec entity contains its own program counter and logical process stack[†]. In ns2, we allocate the smallest split object possible, an instance of `TclObject`, responsible for binding values across the C

[†]Minimum stack size allowed by Parsec is 20 KB.

and Tcl memory spaces. JiST provides the same dynamic configuration capability without requiring the memory overhead of split objects (see section 3.2). (b) – JiST also performs well with respect to event memory requirements. Though they store slightly different data, the C-based ns2 event objects are approximately the same size. On the other hand, Tcl-based ns2 events require the allocation of a new split object per event, thus incurring the larger memory overhead above. Parsec events require twice the memory of JiST events. Presumably,[‡] Parsec uses some additional space in the event structure for its event scheduling algorithm.

The memory requirements per entity and per event in each of the systems are tabulated in Table 3(b). Note that these figures do not include the fixed memory base for the process nor the actual simulation data, thus showing the *overhead* imposed by each approach. Note also that adding simulation data would doubly affect ns2, since it stores data in both the Tcl and C memory spaces. Moreover, Tcl encodes this data internally as strings.

5.4 Performance summary

Having evaluated the computational and memory performance of JiST against ns2, GloMoSim and Parsec, we found that JiST out-performs these systems in both time and space. This section summarizes the important design decisions in each of the systems that bear most significantly on these performance results.

Parsec runs very quickly and there are a number of reasons for this. It is compiled, not interpreted, and uses a modified `gcc` compiler to produce highly optimized binaries. It also uses non-preemptive logical processes to avoid system switching overhead: a Parsec logical context switch is implemented efficiently using only a `setjmp` and a stack pointer update, like a user-level thread switch. The process-oriented model, however, exacts a very high memory cost per entity, since each entity must store a program counter and its stack.

The GloMoSim design compensates for the high per entity overhead of Parsec by aggregating multiple node states into a single entity and inserting an level of indirection in the message dispatch path. While this reduces the number of entities in the system, the indirection comes with a performance penalty. It also eliminates the transparency and many of the advantages inherent to a language-based approach. For example, the aggregation of state

[‡]Could not be validated, since source code was not available.

renders speculative execution techniques impractical.

ns2 is a sequential engine, so message queuing and dispatch are efficient. However, ns2 employs a split object model across C and Tcl to facilitate dynamic simulation configuration. This not only forces a specific coding pattern, it also comes at a performance cost of replicating data between the two memory spaces. More importantly, it exacts a high memory overhead. The ns2 code written in C still runs fast, but the Tcl-based functionality is almost two orders of magnitude slower. Additionally, both ns2 and GloMoSim suffer performance loss from their approaches to simulation configuration that eliminates opportunities for static optimizations, as discussed in section 3.2.

JiST uses a concurrent object model of execution and thus does not require node aggregation. Since entities are objects all within the same heap, as opposed to isolated processes, the memory footprint is small. There is also no context switching overhead on a per event basis and dynamic Java byte-code compilation and optimization result in high computational throughput. Dynamic optimizations can even inline simulation code into the language-based kernel. Since Java is a dynamic language, JiST does not require a split object model for configuration. Instead, we can use reflection to directly observe or modify the same objects used to run the simulation. This both eliminates the performance gap and the additional memory requirements.

6 Design enhancements

Having introduced and evaluated the fundamental JiST system as well as some optimizations to the model, we now discuss various extensions that simplify development and highlight the inherent flexibility of rewriting. We introduce concepts such as blocking events and simulation time concurrency. In the interests of brevity, we omit the discussion of simpler JiST features that are similar to those already commonly found in existing simulation languages, such as statistics gathering and simulation logging, for example.

The ease with which all of these enhancements have been integrated into the basic design underscores the flexibility of the JiST approach and it suggests that JiST is a compelling vehicle for ongoing simulation research. Recent ideas in the simulation literature, such as using reverse computation [11] in optimistic simulation engines and stack-free process-oriented simulation, could readily be implemented within the JiST rewriter.

type safety	-	source and target of event statically checked by compiler
event typing	-	not required; events automatically type-cast as they are dequeued
event structures	-	not required; event parameters automatically marshalled
debugging	-	event dispatch location and entity state available
execution	-	transparently allows for parallel, speculative, and distributed execution

Table 4. Benefits of encoding simulation events as entity method invocations.

6.1 Tight event coupling

The most obvious consequence of the rewriter transformation is that simulation events may be encoded as method invocations, which reduces the amount of simulation code required and improves its clarity without affecting runtime performance. These benefits are summarized in Table 4. The first benefit of this encoding is type-safety, which eliminates a common source of error: the source and target of an event are statically type-checked by the Java compiler. Secondly, the event type information is also managed automatically at runtime, which completely eliminates the many event type constants and associated event type-cast code that are otherwise required. A third benefit is that marshalling of event parameters into the implicit event structures is performed automatically. In contrast, simulators written against event-driven libraries often require a large number of explicit event structures and code to simply pack and unpack parameters from these structures. Finally, debugging event-driven simulators can be onerous, because simulation events arrive at target entities from the simulation queue without any context. Thus, it can be difficult to determine the cause of a faulty or unexpected incoming event. In JiST, an event can automatically carry its context information: the point of dispatch (with line numbers, if source information is available), as well as the state of the source entity. These contexts can then be chained to form an event causality trace, the equivalent of a stack trace. For performance reasons, this information is collected only in JiST's debug mode, but no changes to the application code are required.

The tight coupling of event dispatch and delivery in the form of a method invocation also has important performance implications. Tight event-loops, which can be determined only at runtime, can be dynamically optimized and inlined even across the kernel boundary between JiST and the running simulation, as shown by the Jalapeño project [37]. For example, the dynamic Java compiler may decide to inline portions of the kernel event queuing code into hot spots within the simulation code that frequently enqueue events. Or, conversely, small and frequently executed simulation event handlers may be inlined into the kernel event loop. The tight coupling also abstracts the simulation event queue, which will, in the future, allow the JiST kernel to transparently execute the simulation more efficiently – in parallel, distributed, and even optimistically – without requiring any modifications to the simulation code, as discussed in section 7.1.

6.2 Blocking invocation semantics

We have shown how JiST conveniently models events as invocations on entities. This facility provides all the functionality of an explicit event queue, which is all that many existing simulators use. However, it remains cumbersome to model simulation *processes*, since they must be written as event-driven state machines. While many entities, such as network protocols or routing algorithms, naturally take this event-oriented form, other kinds of entities do not. For example, an entity that models a file-transfer is more readily encoded as a process than as a sequence of events. Specifically, one would rather use a tight loop around a `blocking_send` routine than dispatch `send_begin` events to some transport entity, which will eventually dispatch matching `send_complete` events in return. Many existing applications make use of system calls with blocking semantics. We would like to be able to run such applications within our simulator. To that end, we introduce blocking invocation semantics and *simulation time continuations.*

In order to invoke an entity method with continuation, we merely declare that a given entity method is a *blocking* event. Blocking and non-blocking methods can co-exist within the same entity. Syntactically, an entity method is blocking, if and only if it declares that it throws a `Continuation` exception. This exception is not actually thrown and need not be explicitly handled by a caller. It acts merely as a tag to the rewriter. The semantics of a blocking entity method invocation, as shown in Figure 14, are a natural extension atop the existing event-based invocation. The kernel first saves the call-stack of the calling entity and attaches it to the outgoing event. When the call event is complete, the kernel notices that the event has caller information, so the kernel dispatches a callback event to the caller, with its continuation information. Thus, when the callback event is eventually dequeued, the state is restored and the execution continues right after the point of the blocking entity method invocation. In the meantime, however, local simulation time will have progressed to the simulation time at which

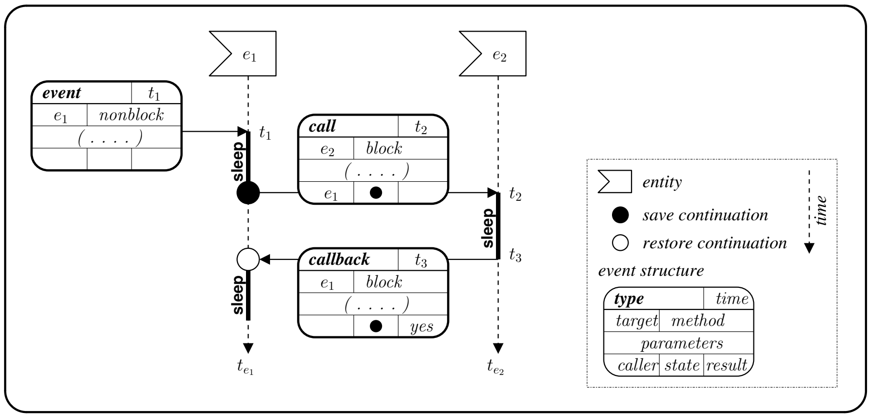


Figure 14. The addition of blocking methods allows simulation developers to regain the simplicity of process-oriented development. When a blocking entity method is invoked, the continuation state of the current event is saved and attached to a call event. When this call event is complete, the kernel schedules a callback event to the caller. The continuation is restored and the caller continues its processing from where it left off, albeit at a later simulation time.

the calling event was completed, and other events may have been processed against the entity.

This approach allows blocking and non-blocking entity methods to co-exist, which allows a combination of event-oriented and process-oriented simulation. Methods can arbitrarily be tagged as blocking, and we extend the basic event structures to store the call and callback information. However, there is no notion of an explicit process, nor even a logical one. Unlike process-oriented simulation runtimes, which must pre-allocate fixed-size stacks for each real or logical process, the JiST stacks are variably-sized and allocated on demand. The stacks are allocated only at the point of the blocking entity invocation, and they exist on the heap along with the event structure that contains it. This reduces memory consumption. Moreover, our model is actually akin to threading, in that multiple continuations can exist simultaneously for a single entity. Finally, there is no system context-switch required. The concurrency occurs only in simulation time, and the underlying events may be executed sequentially within a single thread of control.

Unfortunately, saving and restoring the Java call-stack for continuation is not a trivial task [38]. The fundamental difficulty arises from the fact that stack manipulations are not

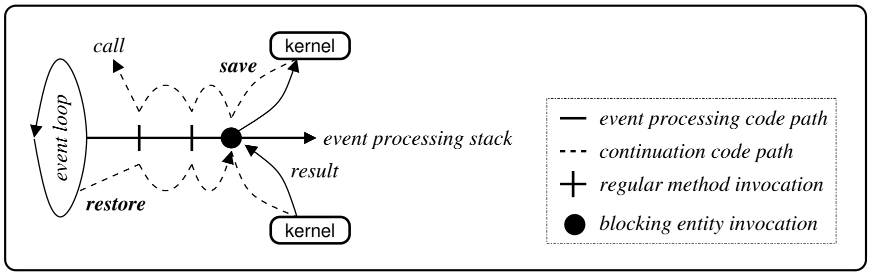


Figure 15. The JiST event loop also functions as a continuation trampoline. It saves the continuation state on a blocking entity method invocation and restores it upon receiving the callback. Due to Java constraints, the stack must be manually unwound and preserved.

supported at either the language, library, or byte-code level. Our solution draws and improves on the ideas in the JavaGoX [39] and the PicoThreads [40] projects, which also save the Java stack for different reasons. Our design eliminates the use of exceptions to carry state information. This is considerably more efficient for our simulation needs, since Java exceptions are expensive. Our approach also eliminates the need to modify method signatures. This fact is significant, since it allows our continuation capturing mechanism to function even across the standard Java libraries. In turn, this enables us, for example, to run standard, unmodified Java network applications within network simulators written atop JiST. A network socket operation is rewritten into a blocking method invocation, so that the application is "frozen" until the network packet is delivered by the simulator.

The rewriter incrementally computes the application call-graph to determine which methods of the simulation code need to be transformed, performs an intra-procedural data-flow analysis to determine the possible execution frame types, and inserts both saving and restoration code around each program continuation location, converting the original simulation program into a continuation-passing style (CPS). The saving code marshals the stack and locals into a custom-generated frame object, containing all the correct field types, and pushes it onto the event continuation stack via the kernel. The restoration code does the opposite and then jumps right back to the point of the blocking invocation.

The kernel functions as the continuation trampoline, as shown in Figure 15. When the kernel receives a request to perform a call with continuation, it registers the call information, switches

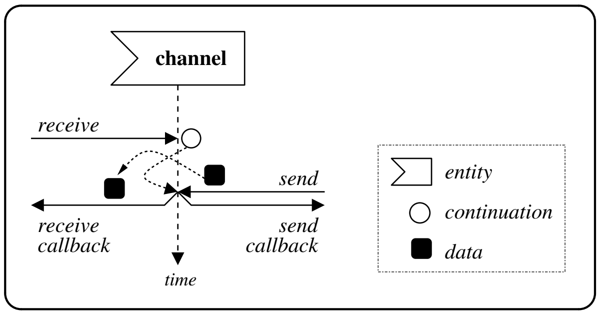


Figure 16. A blocking CSP channel is built using continuations. SWANS uses JiST channels to create simulated sockets with blocking semantics. Other simulation time synchronization primitives can similarly be constructed.

to save mode, and returns to the caller. The stack then unwinds, and eventually returns to the event loop, at which point the call event is dispatched with the continuation attached. When the call event is received, it is processed, and a callback event is dispatched in return with both the continuation and the result attached. Upon receiving this callback event, the kernel switches to restore mode and invokes the appropriate method. The stack then winds up to its prior state, and the kernel receives a request for a continuation call yet again. This time, the kernel simply returns the result of the call event and allows the event processing to continue from where it left off.

The invocation time of a blocking event with this implementation is proportional to the length of the stack. The time to perform a blocking invocation with a short call stack is only around 2-3 times the dispatch time of a regular event. Thus, continuations present a viable, efficient way to reclaim process-oriented simulation functionality within an event-oriented simulation framework. Extensions to the Java libraries and virtual machine that expose the stack in a type-safe manner, as presented in [41], could eliminate this performance gap between non-blocking and blocking events.

6.3 Simulation time concurrency

Using only basic simulation events and continuations, we have built a *cooperative* simulation time threading package within JiST. It can be transparently used as non-pre-emptive replacement for the Java `Thread` class within existing Java applications to be simulated. *Pre-emptive* threading can also be supported, if necessary, by inserting simulation time context switch calls at appropriate code locations during the rewriting phase. However, since we are optimizing for simulation throughput, cooperative concurrency is preferred.

Given simulation time concurrency, one may wish to recreate various simulation time synchronization primitives. As an example, we have constructed the channel primitive from Hoare's Communicating Sequential Processes (CSP) language [42]. It has been shown that other synchronization primitives, such as locks, semaphores, barriers, monitors, and FIFOs, can be constructed using such channels. Or, these primitives can be implemented directly within the kernel. As shown in Figure 16, the CSP channel blocks on the first receive (or send) call and stores the continuation. When the matching send (or receive) arrives, then the data item is transferred across the channel and control returns to both callers. In other words, we schedule two simulation events with the appropriate continuations. A JiST channel is created via the `createChannel` system call. It supports both CSP semantics as well as non-blocking sends and receives. JiST channels are used, for example, within the SWANS implementation of TCP sockets in order to block a Java application when it calls `receive` and to send information back to the application when a packet arrives. In this case, the channel mimics the traditional boundary between the user-level network application and the in-kernel network stack.

7 Discussion

In this section, we turn to a discussion of two directions for possible future work. The first is to extend JiST into a parallel and distributed simulation platform. JiST was explicitly designed with this in mind. Second, we discuss the choice of Java as a platform for JiST and possible improvements to the virtual machine to better support simulation. We also discuss language alternatives and highlight some advantages and disadvantages of the Java choice.

7.1 Parallel, optimistic and distributed simulation execution

The JiST system, as described thus far, is capable of executing simulations sequentially and it does so with performance that exceeds existing, highly optimized simulation engines. JiST also supports *inter*-simulation concurrency. Any number of JiST engines can be started on separate machines, each capable of accepting simulation jobs from a central JiST job queue, where JiST clients post simulations. As each job is processed on the next available server, the corresponding client will service remote class loading requests and receive redirected output and simulation logs. This naïve approach to parallelism has proven sufficient for our needs, since JiST can already model very large networks on individual machines, and it provides us with perfect speed-up for batches of simulations. JiST, however, was explicitly designed to allow concurrent, distributed, and speculative execution, or *intra*-simulation parallelism. By modifying the simulation time kernel, unmodified simulation programs can be executed over a more powerful computing base. These kernels have not been implemented yet. Below we describe the mechanisms that are already in place for such extensions.

Parallel execution in JiST can be achieved by dividing the simulation time kernel into multiple threads of execution, called `Controller`s, which already exist in the single-threaded implementation. Each controller owns and processes the events of a subset of the entities in the system, and controllers synchronize with one another in order to bound their, otherwise, independent forward progress. JiST can further be extended to transparently support entity rollback, so that simulation time synchronization protocols among the various controllers need not be conservative. State checkpoints can be automatically taken through object cloning. Alternatively, efficient undo operators can be statically generated through code inspection in some cases or possibly provided by the developer in other cases for added performance. In any case, entity state changes can always be dynamically intercepted either at the level of complete entities, individual objects, or even individual fields within an object. These state changes can be logged for possible undo, allowing the JiST kernel to transparently perform speculative execution of the simulation.

Controllers may also be distributed in order to run simulations across a cluster of machines. Conveniently, Java support for remote method invocation (or more efficient drop-in alternatives such as KaRMI [43]) combined with automatic object serialization provides location transparency among the distributed controllers. Even beyond this, the existing separator objects (which replace entity references during rewriting) allow entities to dynamically be moved among controllers in the system, for balancing load or for minimizing invocation latency and network bandwidth, while the simulation is running. The automatic insertion of separators between entities provides the simulation developer with a convenient single system image abstraction of the cluster.

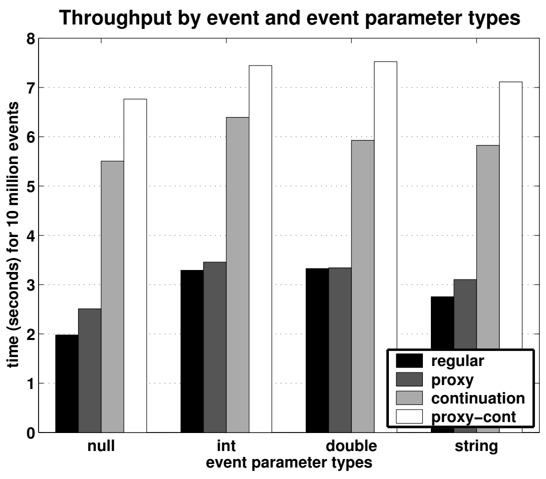


Figure 17. Manually boxing Java primitives for reflection-based invocation and unrolling the Java stack for blocking event continuations are unnecessary Java-related overheads.

7.2 Language alternatives

Given that JiST is a Java-based system, it is natural to question whether Java is an appropriate choice and whether similar benefits could not be attained using other languages. Java has a number of advantages. It is a standard, widely deployed language, not specific to writing simulations. Consequently, the Java platform boasts a large number of optimized virtual machine implementations across many hardware and software configurations, as well as a large number of compilers and languages [44] that target this platform. Java is an object-oriented language and it supports reflection, serialization, and cloning, which facilitates reasoning about the simulation state at runtime. The intermediate byte-code representation conveniently facilitates instrumentation of the code to support the simulation time semantics. Type-safety and garbage collection simplify writing simulations by addressing common sources of error.

Some of these properties exist in other languages as well, and they would be suitable candidates for a JiST-like transformation. Based on our knowledge of existing languages and on our experience implementing JiST, the most suitable candidates include Smalltalk, C#, Ruby, and Python.

Java also has a number of disadvantages that are worth noting, because they adversely affect the performance of JiST. Java primitive types require special handling. They need to be manually "boxed" into their object counterparts for the reflection-based invocation that occurs in the performance-critical event-loop, thus incurring a relatively expensive object instantiation and eventual garbage collection. Instead, the conversion from primitive to object and back should be performed internally by the JVM in an efficient manner, as in C#. Secondly, the Java virtual machine does not support explicit tail call instructions, method calls that can be performed without preserving the stack frame of the caller, since the caller merely returns the result of the callee. These are a common occurrence within an event-based system and adding a byte-code instruction to assist the optimizer could ensure that the proper stack discipline is used. Finally, Java completely hides the execution stack from the programmer. There are many applications that could benefit even from some restricted form of stack access, even as an immutable object. It has been shown that modified virtual machines can support zero-overhead stack access and switching [41]. In JiST, such functionality would eliminate the performance gap between regular and blocking simulation events.

Figure 17 shows the impact of primitive boxing and stack unrolling on various JiST event types. Null events are fastest, since they do not require the instantiation of an argument array. Events with object parameters (`string`) are only slightly faster than events with primitive parameters (`int` and `double`). Proxied events have equivalent performance to regular events, even though they add an additional method invocation. However, this additional stack frame hurts performance in the case of proxied-blocking events, which must disassemble the stack manually, frame by frame. Note also that the proxying method is simply a wrapper, so it is not touched by the CPS transformation. The JVM should certainly implement it using a tail call, if not entirely inline the method.

8 Related work

The JiST and SWANS work spans three domains of research: simulation, networking, and languages. We describe the relevant related work in each of these areas. The interested reader is also encouraged to refer to the doctoral dissertation on this topic [45].

8.1 Simulator construction

Simulation research has a rich history dating back to the late 60s, when it prompted the development of Simula [9]. Many other simulation languages, libraries, and systems have since been designed, focusing on performance, distribution, concurrency, speculative execution, and new simulation application domains.

The first and also most popular approach to building simulators involves the use of simulation libraries. A frequently stressed benefit of this approach is that such libraries are usable within existing general-purpose languages, most often within C or C++. Libraries, such as OLPS [46], Speedes [47], and Yansl [48], provide support for parallel event-driven simulation. SimKit [49] is a simulation class library that supports logical processes. The Compose [8] simulation library allows individual concurrent objects to dynamically adapt their execution modes among a number of conservative and optimistic synchronization protocols.

With the widespread adoption of Java, there has also been research interest in using this language for simulation [50]. SimJava [51] and Silk [52] are two early Java-based libraries for process-oriented discrete-event simulation. However, both of these solutions utilize native Java threads within *each* process or entity to capture simulation time concurrency and therefore do not scale. The IDES library [53] is more reasonably designed, but the project was focussed on using Java for simplifying distributed simulation and did not address sequential performance. Likewise, the Ptolemy II [54] system provides excellent modeling capabilities, but utilizes a sophisticated component framework that imposes overheads in the critical event dispatch path. Finally, the Dartmouth Scalable Simulation Framework (DaSSF) includes hooks to allow for extensions and event handlers written in Java. These projects combine the features of prior simulation library initiatives with the benefits of the Java environment, such as garbage collection, type safety, and portability.

However, regardless of the language chosen, the primary disadvantage of library-based approaches is that the simulation program becomes more complex and littered with simulation library calls and callbacks. This level of detail not only obscures simulation correctness, but also impedes possible high-level compiler optimizations and program transformations. In other words, library-based approaches lack transparency. Noting the advantages of writing simulators in standard language environments, JiST was designed to work within Java. However, JiST provides its simulation functionality using a language-based approach, rather than via a library.

A second approach to building simulators involves the use of simulation-specific languages, which are often closely related to popular existing languages and contain extensions for events, synchronization, and other simulation time primitives. For instance, Csim [55], Yaddes (Parsimony) [56], Maisie [57], and Parsec [10] are all derivatives of either C or C++ that support process-oriented simulation. The resulting executables can be run using a variety of kernels ranging from the sequential to the optimistically parallel.

Simulation language research has also focused on applying object-oriented concepts to simulation-specific problems. Projects, such as Sim++ [58], Pool [59], ModSim II [60], and Rosette [61], have investigated various object-oriented possibilities for checkpointing, concurrency, synchronization, and distribution in the context of simulation. For example, the Moose language [62], a derivative of Modula-2, uses inheritance to specialize implementations of simulation objects with specific knowledge and simulation algorithms to improve efficiency. Languages, such as Act++ [63] and others, structure the concurrency of object-oriented simulations using actors, as opposed to the traditional concurrent object or process-oriented execution models. JiST inherits the object-oriented properties of the Java language and extends the Java object model and execution semantics to support both concurrent object and process-oriented simulation.

Other simulation languages, such as Apostle [64] and TeD [65] have taken a more domain-specific language approach. TeD, for example, is an object-oriented language developed mainly for modeling telecommunications network elements and protocols. OOCSMP [66] is another high-level simulation language designed for *continuous* models expressed as partial differential equations, and is compiled down to Java for efficient execution. While JiST is currently a general-purpose discrete-event simulation platform, the design could certainly subsume domain-specific extensions without loss of generality. In fact, the inherent flexibility of byte-code level rewriter would facilitate this work.

Finally, researchers have built simulations using special operating system kernels that can transparently run processes in virtual time. The landmark work on this topic is the TimeWarp OS [6], and projects such as GTW [67], Warped [68], Parasol [69], and others, have investigated important dynamic optimizations within this model. JiST provides protection at finer granularity by using safe language techniques and eliminates the runtime overhead of process-level isolation.

In summary, JiST merges simulation ideas from both the systems and languages camps by leveraging virtual machines as a simulation platform. To the best of our knowledge, JiST is the first system to integrate simulation execution semantics directly into the execution model of a standard language environment.

8.2 Network simulation

The networking community depends heavily on simulation to validate its research. The ns2 [19] network simulator has had a long history with the community and is widely trusted. It was therefore extended to support mobility and wireless protocols [70]. Though it is primarily used sequentially in the community, researchers have extended ns2 to PDNS [14], allowing for conservative parallel execution. GloMoSim [30] is a newer simulator written in Parsec [10] that has recently gained popularity within the wireless ad hoc networking community. The sequential version of GloMoSim is freely available. The conservatively parallel version has been commercialized as QualNet [33]. Another notable and commercially-supported network simulator is OPNet. Recently, the Dartmouth Scalable Simulation Framework (DaSSF) has also been extended with SWAN[§] [15] to support distributed wireless ad hoc network simulations. Likewise, TeD [65] has been extended with WiPPET [16], though it is focussed on cellular networks. SWiMNet [17] is another parallel wireless simulator focussed on cellular networks.

In this paper, we have described JiST running wireless network simulations of one million nodes on a *single* commodity machine with a 2.2GHz processor and 2GB of RAM. To the best of our knowledge, this exceeds the performance of every existing sequential ad hoc wireless network simulator. Based on the literature, this scale of network is beyond the reach of many parallel simulators, even when using more hardware. Clearly, memory consumption depends on what is being simulated, not just on the number of nodes in the network. For example, in our simulation the state of the entire stack of each node consumes less than 1K of memory, but this will clearly increase if the TCP component is used. Likewise, simulation performance depends on network traffic, node density and many other parameters. Therefore, merely as a point of reference, [29] summarizes the state of the art in 2002 as follows: using either

[§]Not to be confused with our SWANS

expensive multi-processor machines or clusters of commodity machines connected with fast switches, DaSSF, PDNS, and WARPED can simulate networks of around 100,000 nodes, while TeD and GloMoSim have shown results with 10,000 node networks. More recently, the PDNS website [71] states that "PDNS has been tested on as many as 136 processors simulating a 600,000+ node network", but without further details. This same group continues to push the envelope of parallel and distributed simulation further still [72], with GTNetS [73].

Various projects, including EmuLab [74], ModelNet [75] and PlanetLab [76] provide alternatives to simulation by providing emulation and execution test-beds. JiST simulation is complementary to these approaches. However, the ability to efficiently run standard network applications over simulated networks within JiST blurs the distinction between simulation and emulation.

Finally, J-Sim (JavaSim) [77] is a relatively new and highly-optimized sequential network simulator written in Java, using a library that supports the construction of simulations from independent `Component`s with `Port`s that are connected using `Wire`s. The system is intelligently designed to reduce threading overhead, synchronization costs, and message copying during event dispatch, resulting in performance only just slightly slower than JiST. However, the memory overhead for the various JavaSim infrastructure objects, results in a memory footprint that is larger than JiST by an order of magnitude for network models of equal size.

8.3 Languages and Java-related

Java, because of its popularity, has become the focus of much recent research. The Java virtual machine has not only undergone extensive performance work, it has also become the compilation target of many other languages [44]. Projects such as JKernel [21] have investigated the advantages of bringing traditional systems ideas of process isolation and resource accounting into the context of a safe language runtime. The Jalapenõ project [37] has also demonstrated performance advantages of a language-based kernel. JiST makes similar claims in the context of simulation.

A vast number of projects have used Java byte-code analysis and rewriting techniques for a variety of purposes. The Byte-Code Engineering Library [78], Soot [79], and other libraries considerably simplify this task. AspectJ [22] exposes a rewriting functionality directly within the language. Others, including cJVM [80] and Jessica [81], have used Java byte-code rewriting techniques to provide an abstraction of a single-system image abstraction over a cluster of machines. The MagnetOS project [82] has extended this idea to support transparent code migration in the context of an ad hoc network operating system [83]. The JavaParty [84] and the xDU [85] projects have looked at mechanisms to facilitate Java application partitioning and distribution. The JavaGoX [39] and PicoThreads [40] projects among others, have considered the problem of efficiently capturing stack information without modifying the JVM, as proposed in [41]. KaRMI [43] improves RPC performance using a fast drop-in replacement for Java RMI that uses static byte-code analysis to generate specialized marshalling routines. Finally, [23] and [86] have performed static analysis to determine the mutability of Java objects for a variety of optimizations. JiST brings these and other ideas to bear on the problem of high-performance simulation.

9 Conclusion

In this paper, we have proposed a new approach to simulator construction that leverages virtual machines. In particular, we have introduced JiST, a new Java-based simulation framework that executes discrete event simulations both efficiently and transparently by embedding simulation time semantics directly into the Java execution model. We have outlined our rationale for this new design and contrasted it with the existing language-based and systems-based approaches to building simulators. We then evaluated the system, showing that it performs well both in terms of time and memory consumption. We have constructed SWANS, a wireless ad hoc network simulator, atop JiST, as a validation of our approach and have demonstrated that SWANS can scale to wireless network simulations of a million nodes on a single commodity machine. Finally, we have explored the inherent flexibility of the JiST approach by introducing various additional concepts into the JiST model, such as process-oriented simulation, simulation time concurrency primitives, and the ability to run existing and unmodified Java network applications over simulated SWANS networks. We hope that the performance of JiST, its ability to merge ideas from the systems-oriented and the language-oriented approaches to simulation, and the popularity of the Java language will facilitate its broader adoption within the simulation community.

Acknowledgements

The authors acknowledge and thank Benjamin Atkin and Professor Greg Morrisett for their ongoing help and valuable suggestions towards this work. We thank Professors Paul Francis and Alin Dobra for their kind donation of computational resources, as well as Kelwin Tamtoro, Clifton Lin and Ben Viglietta for their work on various SWANS components.

REFERENCES

1. Rimon Barr and Zygmunt J. Haas. JiST/SWANS website, April 2004. `http://jist.ece.cornell.edu/`.
2. Jayadev Misra. Distributed discrete event simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.
3. Richard M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.
4. David M. Nicol and Richard M. Fujimoto. Parallel simulation today. *Annals of Operations Research*, pages 249–285, December 1994.
5. Richard M. Fujimoto. Parallel and distributed simulation. In *Winter Simulation Conference*, pages 118–125, December 1995.
6. David R. Jefferson and et. al. Distributed simulation and the Time Warp operating system. In *ACM Symposium on Operating Systems Principles*, pages 77–93, November 1987.
7. Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *ACM Symposium on Operating Systems Principles*, December 1995.
8. Jay M. Martin and Rajive L. Bagrodia. Compose: An object-oriented environment for parallel discrete-event simulations. In *Winter Simulation Conference*, pages 763–767, December 1995.
9. Ole-Johan Dahl and Kristen Nygaard. Simula, an Algol-based simulation language. *Communications of the ACM*, pages 671–678, 1966.
10. Rajive L. Bagrodia, Richard Meyer, Mineo Takai, Yuan Chen, Xiang Zeng, Jay Martin, and Ha Yoon Song. Parsec: A parallel simulation environment for complex systems. *IEEE Computer*, 31(10):77–85, October 1998.
11. Christopher D. Carothers, Kalyan S. Perumalla, and Richard M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. In *Workshop on Parallel and Distributed Simulation*, pages 126–135, May 1999.
12. Richard M. Fujimoto. Parallel discrete event simulation: Will the field survive? *ORSA Journal on Computing*, 5(3):213–230, 1993.
13. David M. Nicol. Parallel discrete event simulation: So who cares? In *Workshop on Parallel and Distributed Simulation*, June 1997.
14. George Riley, Richard M. Fujimoto, and Mostafa A. Ammar. A generic framework for parallelization of network simulations. In *Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication*, March 1999.

15. Jason Liu, L. Felipe Perrone, David M. Nicol, Michael Liljenstam, Chip Elliott, and David Pearson. Simulation modeling of large-scale ad-hoc sensor networks. In *Simulation Interoperability Workshop*, 2001.
16. Owen Kelly, Jie Lai, Narayan B. Mandayam, Andrew T. Ogielski, Jignesh Panchal, and Roy D. Yates. Scalable parallel simulations of wireless networks with WiPPET. *Mobile Networks and Applications*, 5(3):199–208, 2000.
17. Azzedine Boukerche, Sajal K. Das, and Alessandro Fabbri. SWiMNet: A scalable parallel simulation testbed for wireless and mobile networks. *Wireless Networks*, 7:467–486, 2001.
18. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.
19. Steven McCanne and Sally Floyd. ns (Network Simulator) at `http://www-nrg.ee.lbl.gov/ns`, 1995.
20. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.
21. Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in Java. In *USENIX Annual Technical Conference*, pages 259–270, June 1998.
22. Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. *European Conference on Object-Oriented Programming*, 1241:220–242, 1997.
23. Marina Biberstein, Joseph Gil, and Sara Porat. Sealing, encapsulation, and mutability. In *European Conference on Object-Oriented Programming*, pages 28–52, June 2001.
24. David Anderson, Hari Balakrishnan, Franz Kaashoek, and Robbert Morris. Resilient overlay networks. In *ACM Symposium on Operating Systems Principles*, October 2001.
25. Ion Stoica, Robert Morris, David Karger, Franz Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160, 2001.
26. B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.
27. Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *ACM Conference on Distributed Systems Platforms*, pages 329–350, November 2001.
28. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *SIGCOMM*, 2001.
29. George Riley and Mostafa Ammar. Simulating large networks: How big is big enough? In *Conference on Grand Challenges for Modeling and Sim.*, January 2002.
30. Xiang Zeng, Rajive L. Bagrodia, and Mario Gerla. GloMoSim: a library for parallel simulation of large-scale wireless networks. In *Workshop on Parallel and Distributed Simulation*, May 1998.
31. Valeri Naoumov and Thomas Gross. Simulation of large ad hoc networks. In *ACM MSWiM*, pages 50–57, 2003.
32. Doug Bagley. The great computer language shoot-out, 2001. `http://www.bagley.org/~doug/shootout/`.
33. Qualnet. `http://www.scalable-networks.com/`.
34. Zygmunt J. Haas. A new routing protocol for the reconfigurable wireless networks. In *IEEE Conference on Universal Personal Comm.*, October 1997.

35. David B. Johnson and David A. Maltz. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*. Kluwer Academic Publishers, 1996.
36. David M. Nicol. Comparison of network simulators revisited, May 2002.
37. Bowen Alpern, C. Richard Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Ton Ngo, Mark F. Mergen, Janice C. Shepherd, and Stephen Smith. Implementing Jalapeño in Java. In *Object-Oriented Programming Systems, Languages and Applications*, pages 314–324, November 1999.
38. T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode transformation for portable thread migration in Java. In *International Symposium on Mobile Agents*, 2000.
39. Tatsurou Sekiguchi, Takahiro Sakamoto, and Akinori Yonezawa. Portable implementation of continuation operators in imperative languages by exception handling. *Lecture Notes in Computer Science*, 2022:217+, 2001.
40. Andrew Begel, Josh MacDonald, and Michael Shilman. PicoThreads: Lightweight threads in Java. Technical report, UC Berkeley, 2000.
41. Sara Bouchenak and Daniel Hagimont. Zero overhead java thread migration. Technical Report 0261, INRIA, 2002.
42. C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.
43. Michael Philippsen, Bernhard Haumacher, and Christian Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, 2000.
44. Robert Tolksdorf. Programming languages for the Java virtual machine at `http://www.robert-tolksdorf.de/vmlanguages`, 1996-.
45. Rimon Barr. *An Efficient, Unifying Approach to Simulation Using Virtual Machines.* PhD thesis, Cornell University, May 2004.
46. Marc Abrams. Object library for parallel simulation (OLPS). In *Winter Simulation Conference*, pages 210–219, December 1988.
47. Jeff S. Steinman. SPEEDES: Synchronous parallel environment for emulation and discrete event simulation. In *SCS Multiconference on Advances in Parallel and Distributed Simulation*, pages 95–101, January 1991.
48. Jeffrey A. Joines and Stephen D. Roberts. Design of object-oriented simulations in C++. In *Winter Simulation Conference*, pages 157–165, December 1994.
49. Fabian Gomes, John Cleary, Alan Covington, Steve Franks, Brian Unger, and Zhong Ziao. SimKit: a high performance logical process simulation class library in C++. In *Winter Simulation Conference*, pages 706–713, December 1995.
50. Richard A. Kilgore, Kevin J. Healy, and George B. Kleindorfer. The future of Java-based simulation. In *Winter Simulation Conference*, pages 1707–1712, December 1998.
51. Fred Howell and Ross McNab. SimJava: A discrete event simulation library for Java. In *International Conference on Web-Based Modeling and Simulation*, January 1998.
52. Kevin J. Healy and Richard A. Kilgore. Silk : A Java-based process simulation language. In *Winter Simulation Conference*, pages 475–482, December 1997.
53. David M. Nicol, Michael M. Johnson, and Ann S. Yoshimura. The IDES framework: a case study in development of a parallel discrete-event simulation system. In *Winter Simulation Conference*, pages 93–99, December 1997.

54. S. Bhattacharyya, E. Cheong, J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, B. Vogel, W. Williams, Y. Xiong, and H. Zheng. Heterogeneous concurrent modeling and design in Java. Technical Report UCB/ERL M02/23, UC Berkeley, EECS, August 2002.
55. Herb Schwetman. Csim18 - the simulation engine. In *Winter Simulation Conference*, pages 517–521, December 1996.
56. Bruno R. Preiss. The Yaddes distributed discrete event simulation specification language and execution environment. In *SCS Multiconference on Distributed Simulation*, pages 139–144, 1989.
57. Rajive L. Bagrodia and Wen-Toh Liao. Maisie: A language for the design of efficient discrete-event simulations. *IEEE Transactions on Software Engineering*, 20(4):225–238, April 1994.
58. Dirk Baezner, Greg Lomow, and Brian W. Unger. Sim++: The transition to distributed simulation. In *SCS Multiconference on Distributed Simulation*, pages 211–218, January 1990.
59. Pierre America and Frank van der Linden. A parallel object-oriented language with inheritance and subtyping. In *Object-Oriented Programming Systems, Languages and Applications*, pages 161–168, October 1990.
60. O.F. Bryan, Jr. Modsim II - an object-oriented simulation language for sequential and parallel processors. In *Winter Simulation Conference*, pages 122–127, December 1989.
61. Chris Tomlinson and Vineet Singh. Inheritance and synchronization in enabled-sets. In *Object-Oriented Programming Systems, Languages and Applications*, pages 103–112, October 1989.
62. Jerry Waldorf and Rajive L. Bagrodia. MOOSE: A concurrent object-oriented language for simulation. *International Journal of Computer Simulation*, 4(2):235–257, 1994.
63. Dennis G. Kafura and Keung Hae Lee. Inheritance in Actor-based concurrent object-oriented languages. *IEEE Computer*, 32(4):297–304, 1989.
64. David Bruce. What makes a good domain-specific language? APOSTLE, and its approach to parallel discrete event simulation. In *Workshop on Domain-specific Languages*, January 1997.
65. Kalyan S. Perumalla, Richard M. Fujimoto, and Andrew Ogielski. TeD - a language for modeling telecommunication networks. *SIGMETRICS Performance Evaluation Review*, 25(4):4–11, 1998.
66. Juan de Lara and Manuel Alfonseca. Visual interactive simulation for distance education. *Simulation: Transactions of the Society for Modeling and Simulation International*, 1(79):19–34, 2003.
67. Samir Ranjan Das, Richard M. Fujimoto, Kiran S. Panesar, Don Allison, and Maria Hybinette. GTW: A time warp system for shared memory multiprocessors. In *Winter Simulation Conference*, pages 1332–1339, December 1994.
68. D. E. Martin, T. J. McBrayer, and P. A. Wilsey. Warped: A time warp simulation kernel for analysis and application development. In *International Conference on System Sciences*, pages 383–386, January 1996.
69. Edward Mascarenhas, Felipe Knop, and Vernon Rego. Parasol: A multithreaded system for parallel simulation based on mobile threads. In *Winter Simulation Conference*, pages 690–697, December 1995.
70. David B. Johnson. Validation of wireless and mobile network models and simulation. In *DARPA/NIST Workshop on Validation of Large-Scale Network Models and Simulation*, May 1999.
71. George Riley. PDNS, July 2003. `http://www.cc.gatech.edu/computing/compass/pdns/`.
72. Richard Fujimoto, George Riley, Kalyan Perumalla, Alfred Park, Hao Wu, and Mostafa Ammar. Large-scale network simulation: how big? how fast? In *Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication*, October 2003.

73. George Riley. The Georgia Tech Network Simulator. In *SIGCOMM Workshop on Models, methods and tools for reproducible network research*, pages 5–12, 2003.
74. B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *ACM Symposium on Operating Systems Design and Implementation*, December 2002.
75. Modelnet. `http://issg.cs.duke.edu/modelnet.html`.
76. Planetlab. `http://www.planet-lab.org/`.
77. H.-Y. Tyan and C.-J. Hou. JavaSim: A component based compositional network simulation environment. In *Western Simulation Multiconference*, January 2001.
78. Markus Dahm. Byte code engineering with the BCEL API. Technical Report B-17-98, Freie Universität Berlin, Institut für Informatik, April 2001.
79. Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *CASCON*, pages 125–135, 1999.
80. Yariv Aridor, Michael Factor, and Avi Teperman. cJVM: A single system image of a JVM on a cluster. In *International Conference on Parallel Processing*, September 1999.
81. Matchy J. M. Ma, Cho-Li Wang, Francis C. M. Lau, and Zhiwei Xu. JESSICA: Java-enabled single system image computing architecture. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 2781–2787, June 1999.
82. Rimon Barr, John C. Bicket, Daniel S. Dantas, Bowei Du, T. W. Danny Kim, Bing Zhou, and Emin Gün Sirer. On the need for system-level support for ad hoc and sensor networks. *ACM SIGOPS Operating Systems Review*, 36(2):1–5, April 2002.
83. Rimon Barr, T.W. Danny Kim, Ian Yee Yan Fung, and Emin Gün Sirer. Automatic code placement alternatives for ad hoc and sensor networks. Technical Report 2001-1853, Cornell University, Computer Science, November 2001.
84. Michael Philippsen and Matthias Zenger. JavaParty — Transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.
85. Samir Gehani and Gregory Benson. xDU: A Java-based framework for distributed programming and application interoperability. In *Parallel and Distributed Computing and Systems Conference*, 2000.
86. Igor Pechtchanski and Vivek Sarkar. Immutability specification and its applications. In *Java Grande*, November 2002.