Virtual Machine-based Simulation

Rimon Barr
<barr@cs.cornell.edu>
Wireless Network Lab – Prof. Zygmunt J. Haas
Computer Science, Cornell University

25 March 2003
York University, Toronto

http://www.cs.cornell.edu/barr/repository/jist/
motivation: wireless networks

- **discrete event simulators** are useful and needed
  - in physics, biology, finance, meteorology, etc.
  - e.g., **wireless networks**
    - published ad hoc network simulations
      - lack network **size** - $\sim 500$ nodes; or
      - compromise **detail** - packet level; or
      - curtail **duration** - few minutes; or
      - are of sparse **density** - $<10$/km$^2$; or
      - reduce network **traffic** - few packets per node
        i.e. limited simulation scalability
  - wish to simulate
    - a university campus: 30,000 students
    - the U.S. military: 100-150,000 troops
    - sensor networks, smart dust, Ubicomp
      with **Hundreds of thousands** of cheap wireless devices distributed across the environment

**Simulation scalability is important**
what is a simulation?

- **unstructured simulation**: computers compute
- **time structured**: event-oriented vs. process-oriented

**discrete event simulator is a program that:**
- encodes the simulation **model**
- stores the **state** of the simulated world
- performs **events** at discrete simulation times
- **loops** through a temporally ordered **event queue**
- works through **simulation time** as quickly as possible

**desirable properties of a simulator:**
- **correctness** - valid simulation results
- **efficiency** - performance in terms of throughput and memory
- **transparency** - separate correctness from efficiency:
  - write “simple” program in a **standard** language
  - provide implicit optimization, concurrency, distribution, portability, etc.
how do we build simulators?

systems

• simulation kernels
  • control scheduling, IPC, clock
  • processes run in virtual time
  • e.g. TimeWarp OS, Warped
  🌟 transparency ❌ efficiency

• simulation libraries
  • move functionality to user-space for performance; monolithic prog.
  • usually event-oriented
  • e.g. Yansl, Compose, ns2
  ❌ transparency ✨ efficiency

languages

• generic simulation languages
  • introduce entities, messages and simulation time semantics
  • event and state constraints allow optimization
  • both event and process oriented
  • e.g. Simula, Parsec/GloMoSim

• application-specific languages
  • e.g. Apostle, TeD
  ✨ transparency ✨ efficiency
  ✨ new language

virtual machines
the jist approach

• **JiST – **Java **in** Simulation **Time**
  • converts a virtual machine into a simulation platform
  • no new language, no new library, no new runtime
  • merges modern language and simulation semantics
    • combines systems-based and languages-based approaches
  • result: virtual machine-based simulation

<table>
<thead>
<tr>
<th></th>
<th>kernel</th>
<th>library</th>
<th>language</th>
<th>JiST</th>
</tr>
</thead>
<tbody>
<tr>
<td>transparent</td>
<td>++</td>
<td></td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>efficient</td>
<td></td>
<td>+</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>standard</td>
<td>++</td>
<td>++</td>
<td></td>
<td>++</td>
</tr>
</tbody>
</table>
JiST – Java in Simulation Time

system architecture

1. Compile simulation with standard Java compiler
2. Run simulation within JiST (within Java); simulation classes are dynamically rewritten to introduce simulation time semantics:
   • extend the Java object model and execution model
   • instructions take zero (simulation) time
   • time explicitly advanced by the program: sleep(time)
   • progress of time is dependent on program progress
3. Rewritten program interacts with simulation kernel
jist object model

- program state contained in **objects**
- objects contained in **entities**
  - think of an entity as a simulation component
  - an entity is any class tagged with the `Entity` interface
  - each entity runs at its own simulation time
  - as with objects, entities do not share state
  - akin to a JKernel process in spirit, but without the threads!
jist execution model

- entity methods are an event interface
  - simulation time invocation
  - non-blocking; invoked at caller entity time; no continuation
  - like co-routines, but scheduled in simulation time
- entity references replaced with **separators**
  - event channels; act as state-time boundary
  - demarcate a TimeWarp-like process, but at finer granularity
a basic example

- the “hello world” of event simulations

```java
class HelloWorld implements JistAPI.Entity {
    public void hello() {
        JistAPI.sleep(1);
        hello();
        System.out.println("hello world, " +
                          "time=" + JistAPI.getTime());
    }
}
```

- demo!

<table>
<thead>
<tr>
<th>Java</th>
<th>JiST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack overflow @hello</td>
<td>hello world, time=1</td>
</tr>
<tr>
<td></td>
<td>hello world, time=2</td>
</tr>
<tr>
<td></td>
<td>hello world, time=3</td>
</tr>
<tr>
<td></td>
<td>etc.</td>
</tr>
</tbody>
</table>
JistAPI class is the JiST kernel system call interface
permits standard Java compilation and execution

// used in hello example
interface Entity - tag object as entity
long getTime() - return simulation time
void sleep(long ticks) - advance simulation time

// others, to be introduced shortly
interface Timeless - tag object as timeless
interface Proxiable - tag object as proxiable
Entity proxy(target, intface) - create proxy entity
class Continuation ext. Error - tag method as blocking
void run(type, name, args,...) - run program or script
void runAt(Runnable r) - schedule procedure
void endAt(long time) - end simulation
Channel createChannel() - sim. time CSP Channel
void installRewrite(rewriter) - install transformation
EntityRef THIS - this entity reference
EntityRef ref(Entity e) - get entity reference
// ... and few more
**jist micro-benchmark: event throughput**

<table>
<thead>
<tr>
<th>5x10^6 events</th>
<th>time (sec)</th>
<th>vs. reference</th>
<th>vs. JiST</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference</td>
<td>0.74</td>
<td>1.31x</td>
<td>0.76x</td>
</tr>
<tr>
<td>JiST</td>
<td><strong>0.97</strong></td>
<td>1.97x</td>
<td></td>
</tr>
<tr>
<td>Parsec</td>
<td>1.91</td>
<td>2.59x</td>
<td>1.97x</td>
</tr>
<tr>
<td>ns2-C</td>
<td>3.26</td>
<td>4.42x</td>
<td>3.36x</td>
</tr>
<tr>
<td>GloMoSim</td>
<td>9.54</td>
<td>12.93x</td>
<td>9.84x</td>
</tr>
<tr>
<td>ns2-Tcl</td>
<td>76.56</td>
<td>103.81x</td>
<td>78.97x</td>
</tr>
</tbody>
</table>
jist micro-benchmark: memory overhead

<table>
<thead>
<tr>
<th></th>
<th>per entity</th>
<th>per event</th>
<th>10K nodes sim.</th>
</tr>
</thead>
<tbody>
<tr>
<td>JiST</td>
<td>36 B</td>
<td>36 B</td>
<td>21 MB</td>
</tr>
<tr>
<td>GloMoSim</td>
<td>36 B</td>
<td>64 B</td>
<td>35 MB</td>
</tr>
<tr>
<td>ns2 *</td>
<td>544 B</td>
<td>40 B</td>
<td>74 MB</td>
</tr>
<tr>
<td>Parsec</td>
<td>28536 B</td>
<td>64 B</td>
<td>2885 MB</td>
</tr>
</tbody>
</table>
**SWANS**

- **Scalable Wireless Ad hoc Network Simulator**
  - similar functionality to ns2 and GloMoSim, but...
  - runs **standard Java network applications** over simulated networks
  - can simulate networks of **1,000,000 nodes** sequentially, on a single commodity uni-processor
  - runs on top of **JiST**; SWANS is a JiST application
  - uses hierarchical binning for efficient propagation
  - component-based architecture written in Java

---

![Simulator Stack Diagram]

<table>
<thead>
<tr>
<th></th>
<th>Files</th>
<th>Classes</th>
<th>Lines</th>
<th>Semi</th>
</tr>
</thead>
<tbody>
<tr>
<td>JiST</td>
<td>29</td>
<td>113</td>
<td>14057</td>
<td>3502</td>
</tr>
<tr>
<td>SWANS</td>
<td>85</td>
<td>220</td>
<td>28617</td>
<td>6536</td>
</tr>
<tr>
<td>Other</td>
<td>32</td>
<td>80</td>
<td>6939</td>
<td>2509</td>
</tr>
<tr>
<td>Total</td>
<td>146</td>
<td>413</td>
<td>49613</td>
<td>12547</td>
</tr>
</tbody>
</table>
existing wireless simulators

**ns2** is the gold standard
- written in C++ with Tcl bindings
- created for TCP simulation, modified for wireless networks
- processor and memory intensive
- sequential; max. \( \sim 500 \) nodes
- recently “fixed” for \( \sim 5000 \) nodes

**GloMoSim**
- implemented in Parsec, a custom C-like language
- implements “node aggregation,” to conserve memory
- shown \( \sim 10,000 \) nodes on NUMA machine (SPARC 1000, est. \$300k)

**OpNet** – popular commercial option
- fast, specialized computation
- lack sophisticated execution and also credibility

**custom-made** simulators
- good modeling capabilities
- poor scalability

**PDNS** – parallel distributed ns2
- event loop uses RTI-KIT
- uses fast inter-connect to distribute memory requirements
- shown \( \sim 100,000 \) nodes

**SWAN**
- parallelized and distributed using the DaSSF framework
- similar capabilities to GloMoSim
- shown \( \sim 100,000 \) nodes

**rule of thumb**: extra 10x in scale, using at least 10x hardware and cost
## SWANS components

<table>
<thead>
<tr>
<th>function</th>
<th>implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>application</td>
<td>heartbeat; any Java network application</td>
</tr>
<tr>
<td>transport</td>
<td>UDP; TCP [Tamtoro]</td>
</tr>
<tr>
<td>network</td>
<td>IPv4</td>
</tr>
<tr>
<td>routing</td>
<td>ZRP; DSR [Vigletta]; AODV [Lin]</td>
</tr>
<tr>
<td>link</td>
<td>802.11b; naïve; wired</td>
</tr>
<tr>
<td>placement</td>
<td>random; input file</td>
</tr>
<tr>
<td>mobility</td>
<td>static; random waypoint; input file</td>
</tr>
<tr>
<td>interference</td>
<td>independent, ns2; additive, GloMoSim</td>
</tr>
<tr>
<td>fading</td>
<td>zero; Raleigh; Rician</td>
</tr>
<tr>
<td>pathloss</td>
<td>free-space; two-ray</td>
</tr>
<tr>
<td>propagation</td>
<td>linear scan, ns2; flat binning, GloMoSim; hierarchical binning</td>
</tr>
</tbody>
</table>

![Diagram showing SWANS components]
SWANS performance

- simulation configuration
  - application - heartbeat neighbor discovery
  - field - 5x5km²; free-space path loss; zero fading
  - mobility - random waypoint: v=2-5m, p=10s
  - radio - additive noise; standard power, gain, etc.
  - stack - 802.11b, IPv4, UDP
SWANS performance

Time for 15 minute NDP simulation

<table>
<thead>
<tr>
<th>network size (nodes)</th>
<th>ns2 time</th>
<th>GloMoSim time</th>
<th>SWANS time</th>
<th>SWANS-hier time</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>7136.3 s</td>
<td>81.6 s</td>
<td>53.5 s</td>
<td>43.1 s</td>
</tr>
<tr>
<td>5000</td>
<td>6191.4 s</td>
<td>3249.6 s</td>
<td>4887 s</td>
<td>433.0 s</td>
</tr>
<tr>
<td>50000</td>
<td>27570 KB</td>
<td>47717 KB</td>
<td>49262 KB</td>
<td></td>
</tr>
</tbody>
</table>
SWANS performance

<table>
<thead>
<tr>
<th>t=2m</th>
<th>SWANS-hier</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodes</td>
<td>10,000</td>
</tr>
<tr>
<td>initial memory</td>
<td>13 MB</td>
</tr>
<tr>
<td>avg. memory</td>
<td>45 MB</td>
</tr>
<tr>
<td>time</td>
<td>2 m</td>
</tr>
</tbody>
</table>
performance summary

• **SWANS scalability**
  • can simulate *million node wireless networks*
  • *hierarchical binning* allows linear scaling with network size

• **SWANS is a JiST application**
  • a simulation program written using the “JiST approach”

• **scalability depends on:**
  • **time** – efficient simulation event processing
  • **space** – efficient simulation state encoding
benefits of the jist approach

• more than just performance...
• application-oriented benefits
  • type safety source and target statically checked
  • event types not required (implicit)
  • event structures not required (implicit)
  • debugging dispatch source location and state available
• language-oriented benefits
  • Java standard language, compiler, runtime
  • garbage collection cleaner code, memory savings
  • reflection script-based simulation configuration
  • safety fine grained isolation
  • robustness no memory leaks, no crashes
• system-oriented benefits
  • IPC no context switch, no serialization, zero-copy
  • Java kernel cross-layer optimization
  • rewriting no source-code access required
  • distribution provides a single system image abstraction
  • concurrency model supports parallel and speculative execution
• hardware-oriented benefits
  • cost COTS hardware and clusters
  • portability runs on everything
rewriter

- **rewriter properties**
  - dynamic class loader
  - **no source code access required**
  - operates on application packages, not system classes
  - uses Apache Byte Code Engineering Library (BCEL)
  - allows orthogonal additions, transformations and optimizations

- **rewriting phases**
  - application-specific rewrites
  - verification
  - add entity self reference
  - intercept entity state access
  - add method stub fields
  - intercept entity invocations
  - modify entity creation
  - modify entity references
  - modify typed instructions
  - continuable analysis
  - continuation transformation
  - translate JiST API calls
zero-copy semantics

- **timeless object**: a *temporarily stable* object
  - inferred statically as open-world immutable
  - or tagged explicitly with the *Timeless* interface

- **benefits**
  - pass-by-reference **saves memory copy**
    - zero-copy semantics for inter-entity communication
  - saves memory for common shared objects
    - e.g. broadcast network packets
    - rewrite `new` of common types to `hashcons`
configurability

- **configurability** is essential for simulators
  1. source level reuse; recompilation
  2. configuration files read by driver program
  3. driver program is a scripting language engine

- support for multiple scripting languages by reflection
  - no additional code
  - no memory overhead
  - no performance hit
  - Bsh - scripted Java
  - Jython - Python
  - Smalltalk, Tcl, Ruby, Scheme and JavaScript
simulations using real applications

- using entity method invocations...
  - one can easily write **event-driven** entities.
  - what about **process-oriented** simulation?

- **blocking events**
  - any entity method that “throws” a **Continuation** exception
  - event processing frozen at invocation
  - continues after call event completes, at some later simulation time

- **benefits**
  - no explicit process
  - blocking and non-blocking coexist
  - akin to **simulation time threading**
  - can build simulated network sockets
  - can run standard applications over these simulated sockets
capturing continuations

- mark entity method as blocking: `throws Continuation`
- saving and restoring the stack is non-trivial in Java!
simulation time concurrency

using continuations...

- **simulation time Thread**
  - cooperative concurrency
  - can also support **pre-emptive**, but not necessary
- **simulation time concurrency primitives:**
  - **CSP Channel**: JistAPI.createChannel()
  - locks, semaphores, barriers, monitors, FIFOs, ...
rewriter flexibility

• **simulation time transformation**
  • extend Java object model with entities
  • extend Java execution model with events
  • language-based simulation kernel

• **extensions to the model**
  • **timeless objects**: pass-by-reference to avoid copy, saves memory
  • **reflection**: scripting, simulation configuration, tracing
  • **tight event coupling**: cross-layer optimization, debugging
  • **proxy entities**: interface-based entity definition
  • **blocking events**: call and callback, CPS transformation, standard applications
  • **simulation time concurrency**: Threads, Channels and other synch. primitives
  • **distribution**: location independence of entities, single system image abstraction
  • **parallelism**: concurrent and speculative execution
  • **orthogonal additions, transformations and optimizations**

• **platform for simulation research**
  • e.g. reverse computations in optimistic simulation [Carothers ‘99]
  • e.g. stack-less process oriented simulation [Booth ‘97]
summary

- **JiST – Java in Simulation Time**
  - converts virtual machine into simulation platform
  - merges systems and language-based approaches
  - runs SWANS: Scalable Wireless Ad hoc Network Simulator
  - efficient: both in terms of throughput and memory
  - flexible: timeless objects, reflection-based scripting, tight event coupling, proxy entities, continuations and blocking methods, simulation time concurrency, distribution, concurrency ... serve as a research platform
Virtual Machine-based Simulation

THANK YOU.

http://www.cs.cornell.edu/barr/repository/jist/
simulation time

- actual time
  - standard Java program execution semantics
  - progress of program independent of time

- real time
  - need stronger guarantees on progress
  - progress of program made dependent on time

- simulation time
  - progress of time is dependent on program progress
    - instructions take zero (simulation) time
    - time explicitly advanced by the program: `sleep(time)`
  - simulation event loop embedded in virtual machine
  - rewriter introduces simulation time semantics by
    - extending the Java object model
    - extending the Java execution model
JiST features in SWANS

- **SWANS is a JiST application**
  - **entity invocation** tracking time
    - no context switching; zero-copy; cross-layer optimizations;
    - type-safety; implicit event structures and types
  - **timeless objects** packets
    - saves memory; simplifies memory management
  - **proxy entities** network stack
    - restricts communication pattern; simplifies development
  - **reflection** script-based configuration
    - no memory or performance hit; no additional code
  - **continuations** socket implementations
    - run standard Java network applications over simulated network
tight event coupling

- tight coupling of event dispatch and delivery provides numerous benefits:
  - **type safety**
    - source and target of event statically verified by compiler
  - **event typing**
    - not required; events automatically type-cast as they are dequeued
  - **event structures**
    - not required; event parameters automatically marshaled
  - **debugging**
    - event dispatch location and state are available
  - **execution**
    - transparently allows for parallel, optimistic and distributed execution
distribution and concurrency

- **parallelism** multiple controllers
- **distribution** separators allow migration and provide location independence
- **optimism** check-pointing implicitly supported
proxy entities

- proxy entities relay events to a target
  - possible targets: regular object, proxiable object, entity
  - proxiable: any object tagged with `Proxiable` interface

- benefits
  - equivalent performance: `JistAPI.proxy(target, intfce)`
  - interface-based: does not interfere with object hierarchy
  - mix simulation time invocations with regular invocations
  - provides a capability-like isolation for entities
java deficiencies

- manually need to box Java primitive types
- tail invocations not properly detected
- need API for type-safe stack access
- exceptions are very expensive
hierarchical binning

- simulating signal propagation
  - critical to performance and scalability
  - find radios within a given radius
- prior approaches
  - linear scan
  - flat binning
  - function caching
- hierarchical binning
  - location update: amortized expected constant time
  - neighborhood search: time linear in receivers, $O(\text{result set})$
  - amortized expected asymptotically optimal time
package jist.runtime;

public class JistAPI {

    public static interface Entity {
    }
    public static class Continuation extends Error {
    }
    public static interface Timeless {
    }

    public static long getTime() { ... }
    public static void sleep(long n) { }
    public static void end() { }
    public static void endAt(long t) { }

    public static JistAPI.Entity THIS;
    public static EntityRef ref(Entity e) { ... }

    public static interface Proxiable {
    }
    public static Object proxy(Object proxyTarget, Class proxyInterface) { ... }
    public static Object proxyMany(Object proxyTarget, Class[] proxyInterface) { ... }

    public static final int RUN_CLASS = 0;
    public static final int RUN_BSH = 1;
    public static final int RUN_JPY = 2;
    public static void run(int type, String name, String[] args, Object properties) { }

    public static Channel createChannel() { ... }

    public static void setSimUnits(long ticks, String name) { }

    public static interface CustomRewriter {
        JavaClass process(JavaClass jcl);
    }

    public static void installRewrite(CustomRewriter rewrite) { }
}
example: hello world

```java
import jist.runtime.JistAPI;

class hello implements JistAPI.Entity {
    public static void main(String[] args) {
        System.out.println("simulation start");
        hello h = new hello();
        h.myEvent();
    }

    public void myEvent() {
        JistAPI.sleep(1);
        myEvent();
        System.out.println("hello world, t="
                           + JistAPI.getTime());
    }
}
```
example: scripts

BeanShell – scripted Java

```bash
1 System.out.println("starting simulation from BeanShell script!");
2 import jist.minisim.hello;
3 hello h = new hello();
4 h.myEvent();
```

Jython – Python

```python
1 print 'starting simulation from Jython script!'
2 import jist.minisim.hello as hello
3 h = hello()
4 h.myEvent()
```
example: blocking methods

```java
1 import jist.runtime.JistAPI;
2
3 public class cont implements JistAPI.Entity {
4     
5         public void blocking() throws JistAPI.Continuation {
6             
7                 System.out.println("called at t="+JistAPI.getTime());
8                 JistAPI.sleep(1);
9         }
10     
11     public static void main(String args[]) {
12         
13             cont c = new cont();
14             for(int i=0; i<3; i++) {
15                 
16                     System.out.println("i="+i+" t="+JistAPI.getTime());
17                     c.blocking();
18             }
19     }
20 }
```
rewriter overhead

<table>
<thead>
<tr>
<th>class</th>
<th>Object</th>
<th>Timeless</th>
<th>Entity</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>base size</td>
<td>265438</td>
<td>121987</td>
<td>11358</td>
<td>398783</td>
</tr>
<tr>
<td>total increase</td>
<td>23171</td>
<td>9751</td>
<td>9887</td>
<td>42809</td>
</tr>
<tr>
<td>constant pool</td>
<td>15340</td>
<td>6817</td>
<td>7223</td>
<td>29380</td>
</tr>
<tr>
<td>code, etc.</td>
<td>7831</td>
<td>2934</td>
<td>2664</td>
<td>13429</td>
</tr>
</tbody>
</table>

(5.8%) (5.6%) (63.6%) (3.0%) (3.4%) (2.4%) (23.5%) (3.0%)

Figure 21: Rewriter processing increases class sizes. The figures shown above are the increases in bytes (and as a percentage), from the processing of the complete SWANS code-base. The data is split into the three JiST class categories, showing that the majority of the increase occurs among entity classes and that much of the increase is due only to new constant pool entries.