Thesis defense:

**JiST – Java in Simulation Time**

An efficient, unifying approach to simulation using virtual machines

**Rimon Barr**
<br><barr@cs.cornell.edu>

Wireless Network Laboratory

**Committee:** Prof. Zygmunt Haas (advisor), Dr. Robbert van Renesse, Prof. Ken Birman, and Prof. Alan McAdams.

3 May 2004

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

- cost per MIPS declining
  - e.g. Pentium Xeon:
  - ~10,000 MIPS @ ~$200

- emphasis on computation
  - vs. analytical methods
  - vs. empirical methods

- simulators are useful and needed
  examples –
  - physics: electron tunneling, star collisions, particle dynamics, ...
  - biology: protein folding, disease spread, genetic drift, ...
  - earth science: weather prediction, tectonic modeling, water quality, ...
  - finance: portfolio pricing, statistical arbitrage, risk analysis, ...
  - operations: optimize workflow, supply chain, inventory, pricing, ...
  - CS: performance analysis of networks, processors, heuristics, ...
  - ... take any subject X, and google “X simulation”
motivation: simulation scalability

- **e.g., wireless networks**
- **published ad hoc network simulations**
  - lack network **size** - \(~500\) nodes; or
  - compromise **detail** - packet level; or
  - curtail **duration** - few minutes; or
  - are of sparse **density** - \(<10/\text{km}^2\)
  i.e. limited simulation scalability [Riley02]

- **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 [Jefferson87], Warped [Martin96]
  
  transparency    efficiency

• simulation libraries
  • move functionality to user-space for performance; monolithic prog.
  • usually event-oriented
  • e.g. Yansl [Joines94], Compose [Martin95], ns2 [McCanne95]

  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 [Dahl66], Parsec [Bagrodia98] / GloMoSim [Zeng98]

• application-specific languages
  • e.g. Apostle [Bruce97], TeD [Perumalla98]

  transparency    efficiency  
  new language
virtual machine-based simulation

• Thesis statement:

A virtual machine-based simulator benefits from the advantages of both the traditional systems and language-based designs by leveraging standard compilers and language runtimes as well as ensuring efficient simulation execution through transparent cross-cutting program transformations and optimizations.

• 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>
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 JKernel [Hawblitzel98] process in spirit, 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>
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></td>
<td>0.76x</td>
</tr>
<tr>
<td>JiST</td>
<td>0.97</td>
<td>1.31x</td>
<td>1.97x</td>
</tr>
<tr>
<td>Parsec</td>
<td>1.91</td>
<td>2.59x</td>
<td>3.36x</td>
</tr>
<tr>
<td>ns2-C</td>
<td>3.26</td>
<td>4.42x</td>
<td>9.84x</td>
</tr>
<tr>
<td>GloMoSim</td>
<td>9.54</td>
<td>12.93x</td>
<td>78.97x</td>
</tr>
<tr>
<td>ns2-Tcl</td>
<td>76.56</td>
<td>103.81x</td>
<td></td>
</tr>
</tbody>
</table>
jist micro-benchmark: memory overhead

<table>
<thead>
<tr>
<th>Memory</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 [McCane95] and GloMoSim [Zeng98], 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

---

![Diagram of SWANS simulation stack](attachment:swans_diagram.png)

<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>117</td>
<td>14256</td>
<td>3530</td>
</tr>
<tr>
<td>SWANS</td>
<td>85</td>
<td>220</td>
<td>29157</td>
<td>6586</td>
</tr>
<tr>
<td>Other</td>
<td>32</td>
<td>80</td>
<td>7204</td>
<td>2525</td>
</tr>
</tbody>
</table>

Virtual machine-based simulation
SWANS performance

Time for 15 minute NDP simulation

Memory for NDP simulation

<table>
<thead>
<tr>
<th>$t=15m$</th>
<th>ns2</th>
<th>GloMoSim</th>
<th>SWANS</th>
<th>SWANS-hier</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodes</td>
<td>time</td>
<td>memory</td>
<td>time</td>
<td>memory</td>
</tr>
<tr>
<td>500</td>
<td>7136.3 s</td>
<td>58761 KB</td>
<td>81.6 s</td>
<td>5759 KB</td>
</tr>
<tr>
<td>5000</td>
<td>6191.4 s</td>
<td>27570 KB</td>
<td>3249.6 s</td>
<td>4887 KB</td>
</tr>
<tr>
<td>50000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SWANS performance

Time for 2 minute NDP simulation

<table>
<thead>
<tr>
<th>Network Size (Nodes)</th>
<th>SWANS (hier)</th>
<th>SWANS (scan)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1s</td>
<td>1s</td>
</tr>
<tr>
<td>100</td>
<td>10s</td>
<td>10s</td>
</tr>
<tr>
<td>10,000</td>
<td>1m</td>
<td>1m</td>
</tr>
<tr>
<td>100,000</td>
<td>10m</td>
<td>10m</td>
</tr>
<tr>
<td>1 million</td>
<td>1h</td>
<td>1h</td>
</tr>
<tr>
<td>10^5</td>
<td>5.5 h</td>
<td>5.5 h</td>
</tr>
<tr>
<td>10^6</td>
<td>20 ms</td>
<td>20 ms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t=2m</th>
<th>SWANS-hier NDP simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>nodes</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>

Virtual machine-based simulation
**SWANS performance**

Recent network research results:
- Analyzed **cost of route discovery** in large ad hoc networks.
- Showed that the bordercast protocol performance is **independent of node density**, proven to be optimal.
- Bordercast can improve performance of many existing **flooding-based** routing protocols.
- Optimal bordercast performance at **zone radii = 2 hops**.
- **Cost of zone maintenance bounded**, proportional to network mobility.
- **Aggregating link state zone updates** substantially reduces maintenance overhead.

<table>
<thead>
<tr>
<th>$t=2m$</th>
<th><strong>SWANS-hier ZRP simulation</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>nodes</strong></td>
<td>10,000</td>
<td>100,000</td>
</tr>
<tr>
<td><strong>avg. memory</strong></td>
<td>72 MB</td>
<td>367 MB</td>
</tr>
<tr>
<td><strong>time</strong></td>
<td>4 m</td>
<td>41 m</td>
</tr>
</tbody>
</table>
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,
    cross-cutting program transformations and optimizations
  - 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
zero-copy semantics

- **timeless object:** a **temporally 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**
process-oriented simulation

- 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!
using continuations...

- **simulation time Thread**
  - cooperative concurrency
  - can also support **pre-emptive**, but not necessary

- **simulation time concurrency primitives:**
  - CSP Channel [Hoare78]: `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 [Carothers99]
  - e.g. stack-less process oriented simulation [Booth97]
summary

- **JiST – Java in Simulation Time**
  - prototype virtual machine-based simulation platform
  - merges systems and language-based approaches

<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>

- 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
Thesis defense:

**JiST** – **Java in Simulation Time**

An efficient, unifying approach to simulation using virtual machines

THANK YOU.

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