Simplified Concurrency: A Java Simulation Framework

W. Weiland, R. Weatherly, K. Ring, E. Page, R. Mikula, and F. Kuhl
The MITRE Corporation
7515 Colshire Drive
McLean, VA, USA 22102[1]

ABSTRACT

Discrete-event simulation programs may be written as interacting processes, and many languages, both special-purpose, and general-purpose, support this implementation. Extant frameworks in Java supporting the interacting-processes paradigm use coroutines based on Java threads, which limits the number of processes those frameworks can support. Tortuga is a new framework for interacting-processes simulation in Java that exploits support added to the Jikes Research Virtual Machine for efficient, scalable coroutines. The paper describes the framework, the support added to the Jikes RVM, and presents preliminary performance data.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *Concurrent programming structures*, *Coroutines*, *Frameworks.*

General Terms

Algorithms, Measurement, Performance, Design, Human Factors, Languages.

Keywords

Java, Coroutines, Simulation.

1. INTRODUCTION

Computer-based simulation is an application domain which imposes significant complexity on its developers, while generally demanding high performance. Much of this complexity arises from the need to maintain an explicit model of time, and to represent and manage numerous logically concurrent activities. Historically, simulation has been a fertile area for origination of ideas that have proven broadly useful to computer science. Simula, first proposed in the early 1960's by Dahl and Nygaard [1], introduced the idea of object-oriented programming and provided a cooperative concurrency mechanism known as *coroutines* [2]. While object-oriented programming has seen widespread adoption in simulation, at least within the last decade, use of explicit concurrency in simulation has been far more limited because of lack of general-purpose language support and perceived performance deficiency.

In this article we describe the development and implementation of an efficient coroutine mechanism for simulation. Our mechanism is based on the Java language and the IBM Jikes Research Virtual Machine (RVM). The remainder of this article is organized as follows. Section 2 briefly reviews simulation and coroutines. Section 3 introduces our Java-based Tortuga simulation framework and its support for interacting processes. Section 4 describes the virtual machine extensions to support scalable coroutines. Section 5 presents some preliminary performance benchmark results. Conclusions and areas of future work are given in Section 6.

2. SIMULATION AND COROUTINES

A discrete-event simulation is a program that mimics the behavior of a system whose state changes at discrete points in time. Discrete-event simulations have been organized according to several paradigms: scheduled events, interacting processes, and activity scans [3].

Instead of using language-based concurrency, the use of *scheduled event* mechanisms, provides a single-threaded control flow that is multiplexed across objects (the simulation *entities*). An event, which represents a state change that occurs at a moment in simulated time, is passed to interested entities and may cause an update to their internal state, and may also cause one or more entities to post new events to an event queue. These events will be processed when their corresponding simulated time is reached. Although this is a powerful mechanism, it is extremely low-level and requires fragmenting the internal logic of the entities into event handlers. The complexity of this grows as the semantics of loops, methods, and other high-level control constructs must be implicitly spread over potentially numerous event handlers. This complexity creates a code-understanding and maintenance headache for developers; it is likely that the complexity of event-based simulation has created difficulties for large-scale simulation development programs [4].

By contrast, the *interacting process* paradigm condenses the internal logic of an entity into one or more conceptually parallel processes. The process code closely reflects its actual logic – it has a clearly defined beginning, middle, and end, and may contain arbitrary loops and method calls. The passage of time and waiting for actions performed by other entities, or conditions that may arise in the simulation as a whole, are modeled through explicit, cooperative control constructs that generally involve waiting for other entities to update their states. This greatly simplifies programming, and reduces potential for mistakes in defining and handling events; it also permits more thorough compile-time semantics checking to occur.

[1] The authors' e-mail addresses are: {weiland, weather, keven, epage, rmikula, fkuhl}@mitre.org.

A variety of simulation languages exist that support the interacting process paradigm (Simula, as already noted, and proprietary languages such as SLX [5]). In addition, many general-purpose programming languages, including Java, have been modified to support interacting process simulation (see [6] for a comprehensive list). However, none of these Java-based approaches has so far been able to provide high-performance process-based simulation. The reliance that most of these frameworks place on Java threads translates to limits on numbers of active entities constrained by operating system thread limits – typically, several thousand – which is inadequate to the demands of large-scale simulation efforts.

Outside the simulation domain, several efforts to support thread serialization in Java have been reported, primarily for the purpose of migrating threads. Such mechanisms could be used to support coroutines. The efforts reported in [9], [10] were prototypes implemented on the Sun JVM but not maintained after the end of their projects.

Another way to gain access to the call stack to save a thread's state is to interpret the Java code oneself, *i.e.*, to execute a Java interpreter in one's program. There is a research effort [11] that takes this approach, but is not a complete implementation. The advantage of the interpretation approach is its portability, since it can be run on any JVM. Its disadvantages are speed of execution and difficulty of debugging user code.

3. TORTUGA INTERACTING PROCESSES

Within our Tortuga framework, interacting processes are created by extending class Entity, creating instances of those extended classes, and registering those instances with the simulation executive. Each entity has an agenda method. The agenda is where simulation developers place code to model the proactive aspects of entity behavior. The Tortuga framework provides the usual set of services to allow an entity to sense the simulated environment and control the advance of simulation time.

The simple entity depicted in Figure 1 illustrates the major points in the control of interacting processes. When, at time *t*, an instance of class MyEntity is registered with the simulation executive, the executive invokes the agenda method. Computation at lines 3 and 4 takes place at time *t*. In line 4, the entity requests a delay of 5.0 simulation time units using framework method waitForTime. Method waitForTime transfers control to the simulation executive. After simulation time advances 5.0 time units, the executive transfers control back to line 5 where code at that point experiences a simulation time value $t + 5.0$. When the agenda code is complete, control returns to the simulation executive. The executive then excludes the entity from future scheduling consideration and makes it available for garbage collection.

```
1   class MyEntity extends class Entity {
2     public void agenda() {
3       // time is now t
4       waitForTime(5.0);
5       // time is now t + 5.0
8     }
9   }
```

Figure 1: Simple Entity

The executive and all of the entities are themselves *logical processes* – thread-like constructs that are conceptually concurrent, but operate cooperatively rather than preemptively. These logical processes were initially built on Java threads, but were later adapted to an implementation of coroutines.

Tortuga affords the user four mechanisms for synchronizing entities. These are simulation time, action methods, triggers, and populations. These are a significant part of Tortuga's support for the interacting-processes view of discrete-event simulation.

3.1 Simulation Time

The simplest means of synchronizing the behavior of entities is through simulation time. An entity can request to be suspended until a specified interval of simulation time has gone by. The entity calls waitForTime, specifying the interval, as in Figure 1. The entity's execution is suspended in the call, and the Tortuga executive is free to resume the execution of another entity. When the interval has elapsed the executive will resume the execution of the caller, and the call will return. Simulation time will be at the value it had when the call was made plus the interval.

3.2 Action Methods

An action method is a user-defined method on any Entity subclass that has the side effect of resuming the execution of the entity if it was waiting. An entity invokes an action method on a target entity to interrupt it. If the target entity was waiting in a waitForAction or waitForActionOrTrigger call, the call will return. Note that these wait calls specify a time interval. An action method invocation may cause these calls to return before the interval of simulation time has entirely elapsed.

An action method is distinguished from other methods on an Entity subclass by its name, which takes the form action*X*, where *X* is one or more characters. An action method can have any signature. Action methods are detected, and their behavior implemented, through aspect-oriented programming (AOP). Tortuga uses AspectJ [12], a popular implementation of AOP for Java. The application of AOP to action methods is straightforward: a single pointcut recognizes methods on Entity subclasses whose names begin with "action". An "after" advice informs the executive that an action method has been invoked.

While an entity is waiting for action methods, the invocation of any action method on the entity will cause it to return from the wait call. The wait call returns information that allows the entity to determine which action method was invoked.

3.3 Triggers

An action method invocation represents an interruption or event exogenous to an entity. The Tortuga mechanism for action methods allows the entity to respond to the event and to coordinate its response with its internal events. Triggers allow entities to tie their behavior to the state of the simulation.

A user creates a trigger by creating an implementation of the Tortuga-supplied Trigger abstract class. Specifically, the user implements the condition method, which returns a boolean and represents some predicate about the simulation's state. An entity calls waitForActionOrTrigger, supplying a Trigger instance. The execution of the entity will be resumed, and the wait call will return, when the Trigger's condition method evaluates to true. The executive checks the condition of the trigger at each point

where simulation time can change to determine when to resume execution of the waiting entity.

An entity can wait on several triggers at once; its execution will be resumed when any conditions are true. When waiting on a trigger, an entity can specify a backstop time interval; the entity's execution will be resumed when a trigger condition is true, or the backstop interval has expired.

Triggers and action methods are duals. Any logic expressible using a trigger is expressible using an action method, and *vice versa*. Both mechanisms are provided because some logic is more easily expressed with one mechanism than the other. For example, barrier synchronization can be expressed using a trigger, but the trigger is unnatural.

3.4 Waiting on Actions and Triggers

The method waitForActionOrTrigger will return because a trigger has been satisfied, or any of the action methods defined on the entity have been invoked, or the backstop time interval has expired. The waitForActionOrTrigger returns a WaitResult object that can be queried to determine the reason the method returned. Tortuga provides a tool that analyzes the user's entity code and generates a set of integer constants that correspond to various reasons the wait method might have returned. This allows the user to write a clean switch statement like the following:

```
switch (waitForAction(4.0)) {
   case actionMethodOne:
            info("actionMethodOne invoked");
            break;
   case actionMethodTwo:
            info("actionMethodTwo invoked");
            break;
   case WAIT_TIME_EXPIRED:
            info("backstop interval expired");
            break;
   default:
            System.out.println("Something's wrong...");
            break;
}
```

3.5 Populations

Populations are dynamic sets of entities. When an entity requests creation of a population, it specifies a Java class, Entity or a subclass. The resulting Population object contains references to all currently active entities (entities that have begun execution of their agenda methods and have not finished them) of the specified class or a subclass. Tortuga adjusts the Population as entities enter and leave the simulation.

A new population can be created from an existing one by applying a filter, a user-supplied implementation of interface Filter. The filter contains a predicate that is applied to each of the entities in the original population to see if they belong in the filtered population. The filter can be applied to include or exclude entities. The filtered population is dynamic like the original. The purpose of filtered populations is to allow construction of populations like "the set of Aircraft entities whose present position is within *x* of a given point." Populations can also be joined to create their union.

Populations themselves are not a synchronization mechanism, but they can be used as the basis of various pre-defined triggers, such as size thresholds and the inclusion of a specific entity.

As defined the semantics of populations allow their membership to change at every point that simulation time can change. Entities can come and go, and they may be excluded or included based on applicable filters. Tortuga contains a naïve but correct implementation of populations. It may not be very efficient, as it effectively recalculates a population's membership every time it is asked for the population's size or membership. An efficient implementation of populations might be the subject of further research. [12]

3.6 Efficient Trigger Implementation

An entity can wait for a trigger to be satisfied, *i.e.*, until the trigger's condition method returns true. Tortuga's semantics require that the executive behave as if every trigger on which an entity is waiting is evaluated at each point in simulation execution where simulation time can change. This creates an undesirable computational burden, not because the triggers are expensive to evaluate (experience thus far shows most triggers are simple to compute) but because all the triggers must be enumerated at each point.

This experience prompted a desire for more efficient trigger handling. The goal is to reduce the number of triggers to be enumerated and checked. The approach is to instrument user-supplied trigger code so the simulation executive will be notified when a trigger requires evaluation. Thus when an entity begins waiting on a trigger, the trigger can be ignored until the executive is notified that the trigger's state might have changed.

User-supplied trigger code is instrumented as follows. The user's trigger implementation, in the form of a Java class file, is analyzed to discover what fields in the trigger class or other classes the result of the condition method depends on. If the analysis is successful, the trigger is marked as "instrumented." AspectJ aspects are generated that intercept changes to the fields the trigger depends on. The executive is notified by the aspects when fields have changed; the executive knows then it must check the trigger's condition value.

If the analysis is not successful the trigger is not marked as instrumented.

When an entity begins to wait on a trigger the executive determines if the trigger is instrumented. If not, the executive must check the trigger at each point in execution where simulation may change. Thus uninstrumented triggers remain inefficient. However, the executive can place an instrumented trigger in a set that need not be checked until the executive is notified that the trigger's state may have changed. If most triggers are instrumented, and no notifications are generated for those triggers until their state really has changed, then the overall efficiency is improved.

The kind of analysis undertaken of user-supplied trigger code is called dependence analysis and is widely used in code optimization. It is difficult to perform full dependence analysis on Java because of such features as exceptions and synchronization [14]. However traditional flow analysis [15] yields a great deal. The Tortuga analysis does not attempt a full dependence analysis. It seeks to identify four kinds of dependences:

The trigger depends on a primitive field of the trigger's class.

The trigger is an inner class, and it depends on a primitive field of its enclosing class.

The trigger depends on the state of an instance of a supplied class in the Java Collections framework. The instance is represented by a reference field of the trigger class.

As above, but the trigger is an inner class, and it depends on a reference field of the enclosing class.

These four cases, while certainly not exhaustive, represent most of the cases useful to simulation developers. It is natural to write a trigger class as an inner class of an entity; thus the attention paid to the inner class cases.

The dependence analysis uses algorithms from [15] and tools from the ByteCode Engineering Library [16]. Specifically, the analysis takes the condition method of each trigger class, creates a flow graph, and conducts a dependence analysis (a kind of flow analysis) based on the definition of each Java bytecode instruction. In the course of the analysis if instructions or constructs are encountered that do not conform to the cases listed above, the analysis is abandoned and the trigger is not marked as instrumented.

Preliminary performance results from the instrumented triggers are encouraging. Instrumented and uninstrumented triggers were compared in a simulation of air traffic control (a queuing simulation) where triggers were used to mediate access to constrained runways and ground movements. Figure 2 presents the results, the execution time per flight as the number of flights in the simulation increases. Three times are presented: the time using uninstrumented triggers, the time using instrumented triggers, and the time where all the triggers have been replaced by action methods. The latter is included as the optimal case; for this simulation it is easy to replace the triggers with action methods, with the condition computations undertaken by the action method invoker. The executive then has no triggers to check, so the overhead due to triggers disappears. The time for instrumented triggers is substantially less than for uninstrumented, and is near the optimum. This suggests that, for this simulation, instrumenting the triggers makes the simulation substantially more efficient.

4. VIRTUAL MACHINE EXTENSIONS

A significant component of the effort to make Tortuga efficient and scalable was the implementation of a coroutine mechanism in the underlying Java virtual machine. As mentioned previously, we elected to use Jikes RVM [17]. Jikes RVM offered a number of benefits for our work:

Open source, therefore subject to modification and distribution by us;

Written in Java, which made code comprehension simpler and provided greater assurances of correctness;

High-performance – comparable to existing commercial VM's.

It should be mentioned at the outset that Jikes RVM provides a lightweight threading mechanism similar to "green threads" that provides for some user-space manipulation of execution context, in effect offering stack-swapping without being subject to kernel overhead. This did offer some performance improvement over standard Java thread implementations, but still incurs a memory penalty because of the amount of stack space allocated per thread.

4.1 Lightweight Context Switching

To minimize memory use per logical process (i.e., per simulation entity), we created a mechanism that lets us mark a point on the execution stack (the stack of the one thread that initiated the simulation itself), then swap logical processes in and out by copying pieces of stack above the marked point, implementing what is commonly known as a "cactus stack". As entity methods execute, they push activation records onto the stack. When an entity has completed an update of its state at a particular point in time, it calls one of the wait methods described earlier, which in turn yields to the executive's logical process. At this point, the execution stack above the marked point is copied to a buffer, and stored with the logical process object, together with stack pointer, instruction pointer, and registers. The executive's object stack segment is then copied over top of the previous stack context (at the marked point), and the VM's stack pointer, instruction pointer and registers are restored to resume execution from the point the logical process last yielded. The executive determines from a queue of pending waits posted by other logical processes (in fact, an event queue) which logical process is due to be scheduled next, and yields to that process.

4.2 Implementation

Implementation of this "stack swapping" within Jikes RVM proceeded in two phases. The first involved the manipulation of process state (storing and loading of stack segments); the second involved making extensions to the garbage collection system that would enable references stored in these dormant stack segments to be appropriately traced so that references from dormant logical processes would not be released by the garbage collector.

The implementation of state save and restore involves a class called StackSegment; each logical process holds a reference to a StackSegment that is used to track information about the Logical process, including whether its state has currently been saved, and if so, a buffer that holds a copy taken from the top of the stack of the Logical process execution context. This buffer is dynamically sized to accommodate varying stack depths. In the current implementation, StackSegments are explicitly created and discarded as corresponding Logical processes get started and terminated. A StackSegmentManager maintains a list of segments; discarded segments are removed from the list at the next garbage collection, at which point they and associated buffers are freed.

During garbage collection, an additional call is made into the StackSegmentManager to perform a scan of segments. This involves using the garbage collector's normal stack scanning method for marking object and code references on a thread stack, but modified to permit passing an offset value. The offset allows the thread scanning to treat the saved stack buffer as if it had been moved back to its proper place on the stack during the scan, so that addresses of object and code references can be identified and accumulated as base registers for the garbage collector. Normal thread scanning simply uses an offset of zero.

4.3 Impact on Jikes RVM Code

From the standpoint of distributing and maintaining Tortuga modifications to Jikes RVM, it is worth considering the scope and impact of the code changes. The changes fall into several categories:

> Relatively independent extensions, consisting of new Java classes;
>
> General purpose changes in internal APIs and implementations that enable Tortuga support;
>
> Tortuga-specific functional or structural changes to Jikes RVM;
>
> Configuration changes.

In the first category, the Tortuga framework uses three classes to implement coroutine support: Coroutine, which provides methods to perform context switches and an entry point for garbage collection; StackSegment, which is used to maintain logical process (i.e., coroutine) state and support garbage collection; and StackSegmentManager, which manages allocation, disposal, and garbage collection of references from StackSegments. These classes comprise the bulk of the coroutine implementation under Jikes RVM, and consist of fewer than 1,000 lines of code.

StackSegments hold buffers that contain the actual copies of stack contents needed to save and restore coroutine state; these buffers must be scanned at garbage collection time to identify and mark Objects and code that were referenced on the stack. In effect, these partial stack copies can be treated like the "live" execution stack at garbage collection time by extending the functionality of the existing stack scanning code to accept an offset value. This offset represents the distance between the original top of the stack and the beginning of the saved buffer in memory. Since the buffer contains direct copies of intra-stack references and addresses, the stack scanning code needs to offset these references to point to addresses within the buffer. This is the single case of a general-purpose enabling extension to Jikes RVM code. This change affects ScanThread (part of the mmtk memory management subsystem) and the various GCMapIterator implementations. Steve Blackburn, Daniel Frampton, and Robin Garner of ANU were instrumental in making these enhancements, which are now part of the official Jikes RVM code base.

Several relatively minor changes connected our extensions and the existing code. The Coroutine and StackSegmentManager need to be part of the VM boot image so appropriate static references to them were added to the top-level Dummy class that triggers compilation of boot image classes. In addition, at garbage collection time, we need to arrange that the thread scanning mechanism ("ScanThread") causes scanning of the dormant coroutines via the Coroutine class. This requires the addition of a single call to the ScanThread class. It is critical that the saved stack buffers not be relocated as a result of garbage collection; this would cause subsequent adjustment of their internal references to fail. Consequently, memory allocation must be adjusted to allocate these buffers in non-moving space.

Finally, there are a small number of configuration changes, notably in the jconfigure script, that ensure that needed classes correctly appear in the boot image.

Because of the minimal interaction of Tortuga coroutine code with the base Jikes RVM code base, we expect that distributing our changes as a small patchset will be a viable approach (especially considering that thread scanning support has been embedded within the Jikes RVM code base at the time of this writing). The pieces of Jikes RVM that Tortuga interacts with are fairly stable, and the actual intrusive changes are minimal, so that we expect maintenance of Tortuga coroutine support should be feasible on an ongoing basis with limited effort.

5. PERFORMANCE BENCHMARKS

The graph in Figure 2 is a sample of output from the performance benchmark database constructed as part of the Tortuga project. The developers have written a series of benchmark programs to test performance of various aspects of the system. These benchmarks write their results as XML documents to a central XML database. A user can query the database, via web browser, for all reports of a given benchmark, and request that certain reports be plotted.

Several benchmarks were written specifically to test performance as a function of number of entities. These benchmarks typically were limited to a maximum of approximately 6,000 entities on Windows and Linux platforms. Beyond that point, operating system thread limits generally resulted in the system becoming unresponsive. However, using our coroutine extensions to Jikes RVM, it was possible to run these benchmarks with over 100,000 entities with at least equivalent speed. Figure 3 shows corresponding benchmarks for the "Time Advance Round Robin" benchmark. The ThreadFlow curve shows performance as a function of number of entities for an implementation based on Java Threads, while the JikesFlow curve shows the corresponding performance for a coroutine-based implementation.

6. CONCLUSIONS

The Tortuga research program has resulted in a productive framework for simulation developers that lessens the burden of managing synchronization in a concurrent programming domain, while avoiding the contortions required to implement simulations in a single-threaded, event-based approach. Through relatively small, localized additions to the Jikes RVM code, we were able to provide specialized support for coroutines that boosted the performance of our framework to a level suitable for large-scale simulations (hundreds of thousands to potentially millions of entities, with concomitant high computational demands).

The use of coroutines as an underlying concurrency mechanism provides the expressive advantages of concurrency (permitting the solution space code to reflect the problem space form), while relieving the programmer of many concerns relating to locking and race conditions. Our implementation of coroutines in Jikes RVM is subject to extension for use as a general-purpose mechanism. As proposed by Conway [2], coroutines have broader applicability, including common producer-consumer application structures.

The efforts we have described have provided a basis for a scalable, functionally complete, Java-based simulation framework. However, there are a number of areas where this work can be extended.

> **Storage Management**. The current implementation of coroutines under Jikes RVM uses standard Java data

structures that are scanned and collected just like any application data. However, the repeated scanning and rescanning of these buffers for code and object references is likely to be a cause of overhead during garbage collection. If unmodified buffers can have their metadata (root lists) cached between garbage collections, it may be possible to reduce this scanning overhead for many applications.

Serialization. Serialization of coroutine state would provide both for saving and restoring simulation state, and migrating simulations or entities across machines. Serialization may support other types of applications, such as web applications, that can benefit from the structural simplicity that coroutines can provide, but that require persistence. Coroutine migration does not directly enable parallel computation, since the coroutine model does not provide the necessary access controls to shared state; however, in restricted domains (such as simulation), extending our framework to mediate state access between entities could enable some overlapped computation and capitalize on parallel hardware.

Generalization of Coroutine Support. The coroutine support we have built into Jikes RVM requires registering a method whose frame marks the bottom of the stack; this makes our coroutines somewhat specialized to our application and is designed to be used indirectly through Tortuga synchronization APIs (as described in Section 3 above). This coroutine support should be extended to provide more than one coroutine "group", should potentially support nesting of coroutines (multiple branches of the cactus stack), and perhaps to directly support symmetric coroutines. This has potential benefits to a broad range of applications, such as web applications.

We should note that Jikes RVM provided a high-quality platform for development of the experimental coroutine support for Tortuga. Apart from our own learning curve with the code base, it was possible for us to implement coroutine support with reasonable effort and minimal code (around a thousand lines). This is a testament to the design of Jikes RVM, and to the wisdom of the decision to implement its runtime system in Java. We believe that concurrency constructs such as coroutines in Java will prove useful to developers in general, and have some hope that the major suppliers of Java tools will see the wisdom in supporting such techniques in future releases.

7. ACKNOWLEDGMENTS

Our thanks to Dr. Steve Blackburn of the Australian National University for invaluable help in understanding and modifying the Jikes RVM, and to his graduate students Daniel Frampton and Robin Garner for their in-depth assistance in tracking down a host of complex context-management issues.

8. REFERENCES

[1] Dahl, O., and Nygaard, K. SIMULA--An ALGOL-Based Simulation Language. CACM, 9(1966), 671-678.

[2] Conway, M.E. Design of a Separable Transition-Diagram Compiler. CACM 6 (1963), 396-408.

[3] Balci, O. (1988) The Implementation of Four Conceptual Frameworks for Simulation Modeling in High Level Languages. In *Proceedings of the 1988 Winter Simulation Conferenc*e, pp. 287-297 (San Diego, CA, Dec 12-14).

[4] Joint Simulation System (JSIMS). Available online via <www.jfcom.mil/about/fact_jsims.htm> Visited July 14 2004.

[5] Henriksen, J. An Introduction to SLX. In *Proceedings of the 27th Winter SimulationConference (Wintersim 1995)* (Arlington, Viriginia, 1995)

[6] Weatherly, R., and Page, E. Efficient Process Interaction Simulation In Java: Implementing Co-Routines Within A Single Java Thread. In *Proceedings of 36th Winter Simulation Conference (Wintersim 2004).*

[7] Vanek, L. I. and Marty, R. 1979. Hierarchical coroutines a mechanism for improved program structure. In *Proceedings of the 4th international Conference on Software Engineering* (Munich, Germany, September 17 - 19, 1979). International Conference on Software Engineering. IEEE Press, Piscataway, NJ, 274-285.

[8] Ana Lucia de Moura, Roberto Ierusalimschy. Revisiting Coroutines. Technical report MCC15/04, Computer Science Department, PUC-Rio, June 2004.

[9] Bouchenak, S., Hagimont, D., Krakowiak, S., De Palma, N., Boyer, B., Experiences Implementing Efficient Java Thread Serialization, Mobility and Persistence. INRIA Report no. 4662. December 2002.

[10] Weyns, D., Truyen, E., Verbaeten, P., Distributed Threads in Java. In *International Symposium on Distributed and Parallel Computing (ISDPC 2002).*

[11] Jacobs, P., Verbraeck, A., Mastering D-SOL: A Java based suite for simulation. Available online via <www.simulation.tudelft.nl/dsol/dsol/documentation.html> Visited July 14 2004.

[12] Colyer, A., Clement, A., Harley, G., and Webster, M. *Eclipse AspectJ : Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools.* Addison-Wesley, Reading, MA, 2004.

[13] Iwerks, G., Samet, H., Smith, K. Maintenance of Spatial Semijoin Queries on Moving Points. In *Proceedings of the 13th International Conference on Very Large Databases (VLDB 2004)* (Toronto, Canada, August 31 - September 3 2004). Morgan Kaufman, San Francisco, CA.

[14] Chambers, C., Pechtchanski, I., Sarkar, S., Serrano, M., Srinivasan, H. Dependence analysis for Java. In *12th International Workshop on Languages and Compilers for Parallel Computing* (August 1999).

[15] Aho, A., Sethi, R., Ullman, J. *Compilers Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1986.

[16] ByteCode Engineering Library, Available online via <http://jakarta.apache.org/bcel/>. Visited July 2005.

[17] Alpern, B., Augart, S., Blackburn, S.M., Butrico, M., Cocchi, A., Cheng, P., Dolby, J., Fink, S., Grove, D., Hind, M., McKinley, K.S., Mergen, M, Moss, J.E.B., Ngo, T., Sarkar, V., and Trapp, M.. The Jikes Research Virtual

Machine project: Building an Open-Source Research Community. *IBM Systems Journa*l, Vol 44, No 2, 2005.

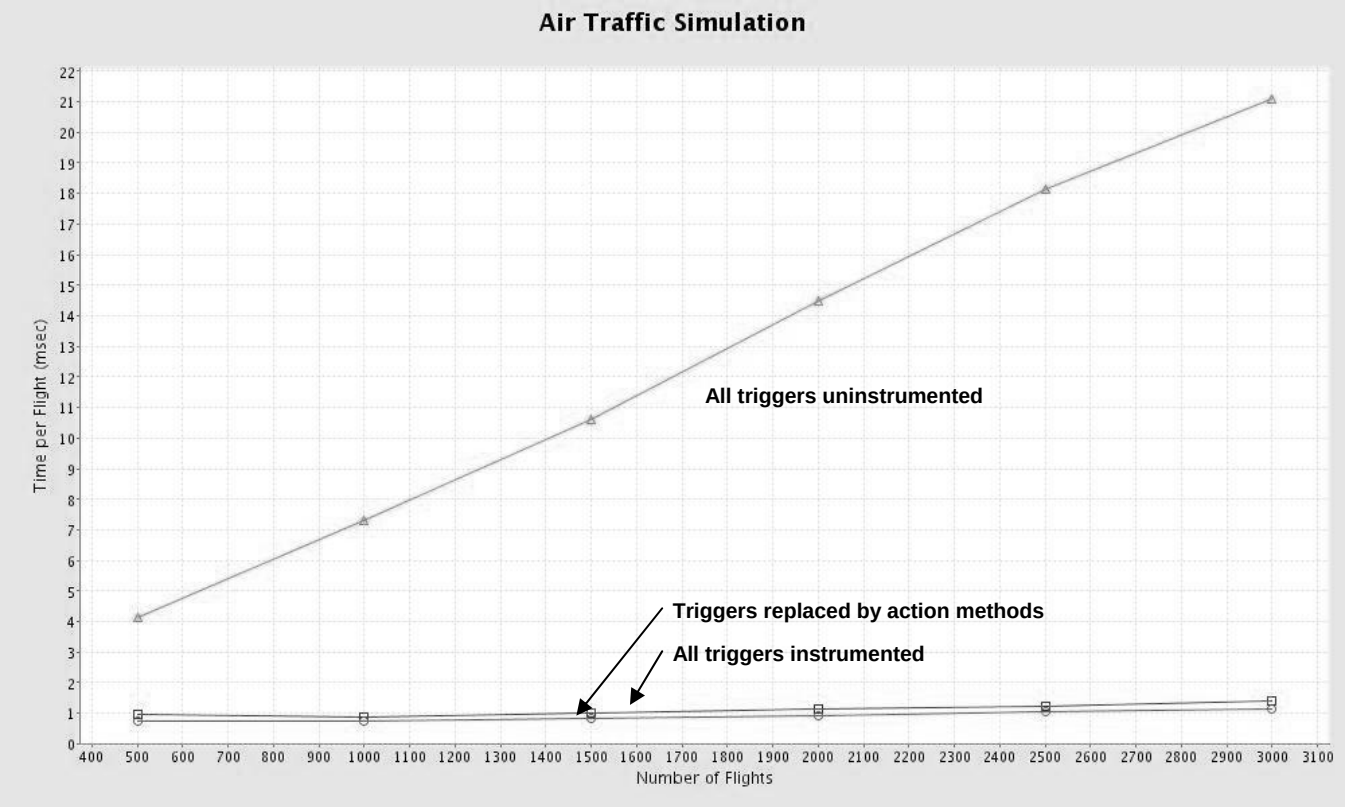

Figure 2. Performance of Instrumented Benchmarks

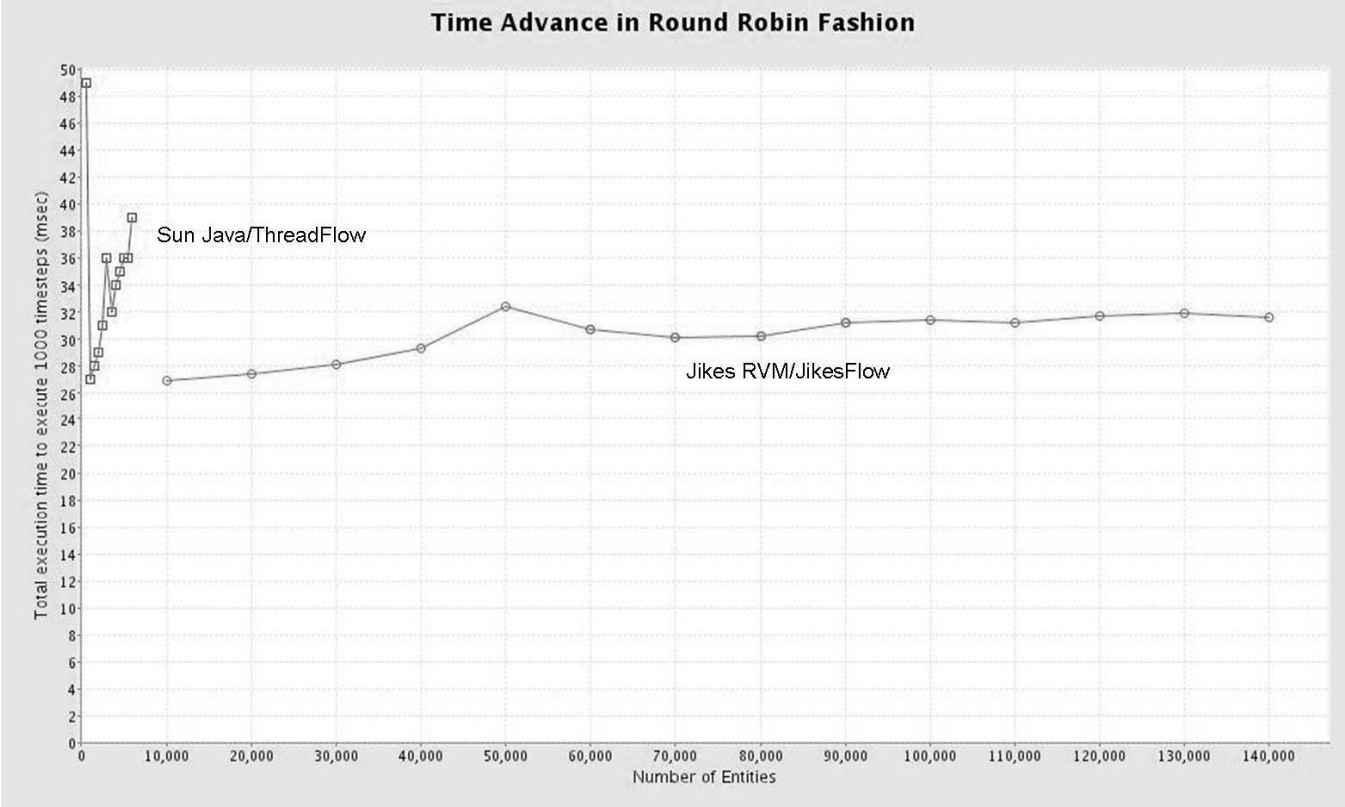

Figure 3. Comparison of Thread and Coroutine Implementations