Debugger Support for Single-Threaded Event-Driven Applications

Fred Reiss and Elaine Cheong

May 2001
CS262B Spring 2001

Abstract

For some applications, message-oriented systems have a performance advantage over process-oriented systems. However, message-oriented systems can be more difficult to debug, due to their unpredictable control flow. Our project allows programmers to debug a message-oriented system using the more intuitive control flow of a process-oriented system. We have developed an algorithm, `threadsim`, that maps sequences of messages and handler functions in a message oriented system onto a simulated process tree in real time with constant amortized overhead per message. Using this algorithm, we have created an extension to `gdb` that can simulate threads on any single-process event-driven system. We prove that our algorithm will work for similar systems that run an arbitrary number of message handlers in parallel.

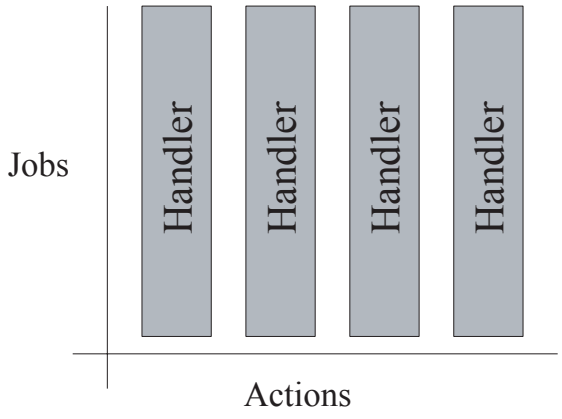

Figure 1: General layout of a message-oriented system. The system performs a set of **jobs**, each of which consists of a set of **actions**.

1 Introduction and Motivation

1.1 Message-Oriented and Process-Oriented Systems

Since the early days of time-sharing and interactive computer programs, programmers have generally structured large interactive software systems in one of two ways. An interactive software system generally has a set of **jobs** that it must perform, each of which consists of a set of **actions**. A message-oriented system (see Figure 1), also known as an event-driven system, is divided into components, or **handlers**, each of which is responsible for one or more actions. To perform a job, the handlers communicate with each other by passing messages. A process-oriented design (see Figure 2) divides the system into processes, each of which handles all the actions for a single job. Processes in a process-oriented system negotiate access to shared resources using locks.

The distinction between a message-oriented system and a process-oriented system was first formal-

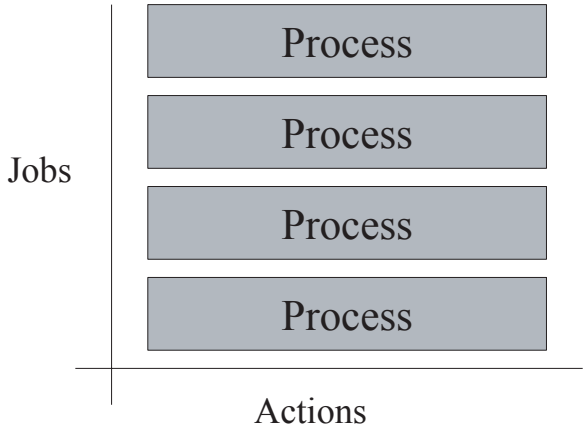

Figure 2: General layout of a process-oriented system. The system performs a set of **jobs**, each of which consists of a set of **actions**.

ized in Lauer and Needham's 1978 paper, "On the Duality of Operating System Structures" [1]. Lauer and Needham noted that there exists a simple direct mapping between their canonical forms of message-oriented and process-oriented systems. Sending a message in a message-oriented system, for example, is equivalent to acquiring a lock in a process-oriented system. Given a message-oriented program, Lauer and Needham presented a simple transformation to an equivalent process-oriented system, and vice versa.

1.2 Performance Advantages of Message-Oriented Systems

In principle, Lauer and Needham asserted, a message-oriented system should have the same performance as its dual process-oriented system. The primitives required to implement the two models are similar, and it should be possible to implement one set of primitives with the same level of efficiency as the other set.

In practice, several researchers [2] [3] [4] have found that a message-oriented design can give significant performance advantages for highly-concurrent network applications. Message-oriented systems allow certain optimizations that can be difficult or impossible to implement for a process-oriented system. A message-oriented system can pass messages in batches, whereas only one process in a process-oriented system can hold a lock at a given time. On today's production operating systems, a message-oriented design give programmers greater flexibility to schedule actions than does a process-oriented system. This flexibility allows programmers to avoid the overhead of priority scheduling and to schedule actions in a way that reuses data in the processor's cache [5]. Architectures that allow programmers to customize the scheduling of threads within a process [6] [7] [8] have not yet found their way into mainstream operating systems. For distributed systems, moving a thread of control from one machine to another using remote procedure calls or thread migration results in significant overhead. As a consequence, giant-scale Internet services [9] typically use a message-oriented architecture.

1.3 Debugging Message-Oriented Systems

In spite of their performance advantages, message-oriented systems can be more difficult to debug than the equivalent process-oriented systems [3]. Often, a programmer wishes to troubleshoot the actions that comprise a single job. For example, someone debugging a Web server may want to know why a certain HTTP request causes the server to crash under heavy load. Unfortunately, the handlers in a message-oriented system typically execute in separate threads of control. Thus, the flow of control in message-oriented systems passes rapidly from one job to another, and a programmer will quickly wander far from the job of interest as she single-steps through such a system. In a process-oriented system, on the other hand, the flow of control follows the logical causal relationships between the actions in a job.

1.4 Overview of our Project

Our project gives programmers the freedom to debug message-oriented systems using the more intuitive control flow of a process-oriented system. We accomplish this goal by creating **simulated threads of control** on top of a message-oriented system. Each simulated thread consists of a set of actions from a single job that are linked together by messages. We have modified the `gdb` debugger [10] to use these simulated threads as it would use conventional operating system threads.

For our initial implementation, we have chosen to focus on a particular type of message-oriented system. A **single-process event-driven** design, also known as a design that follows the Reactor pattern [11], consists of a single component, known as the **event loop**. The event loop communicates with the operating system by using a system call such as `select(2)` to determine whether any interesting I/O **events** have occurred on a set of file descriptors. When an event, such as data arriving on a network port, occurs, the event loop executes the appropriate **handler function**. [2] provides a more detailed description of single-process event-driven systems and their relationship to other types of message-oriented systems.

1.5 Design Goals

Our project had the following overall design goals:

- Our project should require minimal modifications to the software that dispatches events.
- Our project should be extendible in a straightforward way to support multiple event loops, as well as recursive calls into a single event loop.
- Our project should have a minimal performance impact while the system being debugged is not

running inside a debugger, while still permitting the debugging of core dumps.

- Our project should work with the existing graphical front ends to `gdb` without modifications to those graphical front ends.

1.6 Roadmap

The remainder of this paper is organized as follows:

- Section 2 describes the algorithm that our project uses to map events to simulated threads. In sections 2.6 and 2.7, we prove that this algorithm is correct and runs in time proportional to the number of events.
- Section 3 describes the current implementation of our project.
- Section 4 describes several possible future directions for our project.
- Section 5 summarizes related work.
- Section 6 analyzes the extent to which our project met its original design goals.

2 Mapping Events to Simulated Threads

In this section, we describe the algorithm, `threadsim`, that we use to map sets of events to simulated threads. `threadsim` works by maintaining a "process tree" of simulated threads. In this process tree, simulated thread A is the parent of simulated thread B if and only if an event handler in thread A scheduled the first event in thread B. The algorithm allows an arbitrary number of event handlers to run at the same time and adds an overhead proportional to the number of events that occur.

2.1 Definitions

2.1.1 Events

We define an **event** to be an ordered triple $(State, Children, Node)$, where:

- $State : Exp$ is the **state** of the event.
- $Children$ is an ordered list of the events that are **direct descendants** (See section 2.1.3) of the event.
- $Node$ is the Node (See section 2.2) associated with the event.

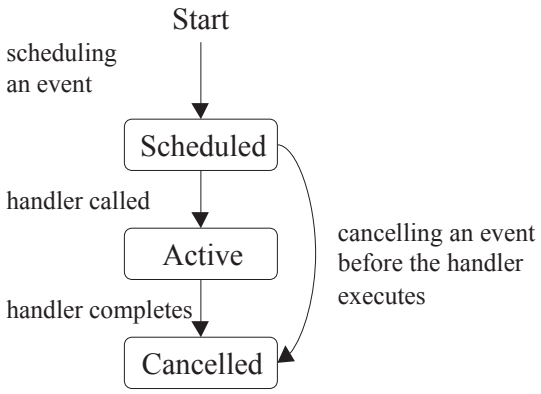

Figure 3: Legal state transitions for an event.

2.1.2 States of Events

At any time, an event can have one of the following four states:

- *Start*, which means that the system has not done anything with regard to the event.
- *Scheduled*, which means that the system has registered an event handler for the event, but the event has not occurred.
- *Active*, which means that the event's handler is executing.
- *Canceled*, which means that the event was registered with the system but is no longer registered.

An event is **alive** if it is in either the *Scheduled* state or the *Active* state.

An event may change from one state to another. The following state transitions are considered **legal**:

- $(Start, Scheduled)$
- $(Scheduled, Active)$
- $(Active, Canceled)$
- $(Scheduled, Canceled)$

Figure 3 summarizes the legal state transitions.

Any state transition that is not legal is an **illegal** state transition.

2.1.3 Properties of Events

Event a is an **unnamed** event if $a.Node.Name =$ `nil`.

Event a is a **named** event if it is not unnamed.

We say that event b is a **direct descendant** of event a if the event handler function for a scheduled event b.

Event b_n is an **indirect descendant** of event a_1 if there exist events $a_2, ..., a_{n-1}$, such that a_{i+1} is a direct descendant of a_i, for all integers i in the range $[1, n-1]$.

Event b is a **rightmost direct descendant** of event a if:

- b is a direct descendant of a, and
- b was the **rightmost** unnamed entry in a's *Children* list when a's state changed from `Active` to `Canceled`.

Event b_n is a **rightmost indirect descendant** of event a_1 if there exist events $a_2, ..., a_{n-1}$, such that a_{i+1} is a rightmost direct descendant of a_i, for all integers i in the range $[1, n-1]$.

2.2 Nodes

The `threadsim` algorithm operates on a "process tree" of simulated threads and a mapping from events to nodes of this tree. Each node in the simulated process tree has the following properties:

- *Parent*, which points to the node's parent node.
- *RightSib* and *LeftSib*, which link the named or unnamed children of the parent node into a list.
- *RightNamedChild* and *RightUnnamedChild*, which point to the rightmost elements of lists of the node's named and unnamed children, respectively.
- *Name* :, the **name** of the node, is the thread ID of the simulated thread that corresponds to the node.

Node N is **unnamed** if $N.Name =$ `nil`.

The root of the tree is a special node, which we denote by *ROOT*. *ROOT* has only named children.

2.3 Naming Conventions

In the sections that follow, we use **uppercase** letters to refer to nodes and **lowercase** letters to refer to events.

We denote the set of all non-root nodes in the simulated process tree by N.

2.4 Invariants

In this section, we outline the invariants that the `threadsim` algorithm maintains.

At any time, all the nodes in N are part of a tree rooted at *ROOT*.

For any node $A \epsilon N$, there is exactly one event e such that $e.Node = A$. This event e is in either the *Scheduled* or the *Active* state. Furthermore, if event e is in the *Scheduled* or *Active* state, then $e.Node$ must not be `nil`.

Consider an arbitrary node $A \epsilon N$. Let a be the event such that $a.Node = A$. Let B be $A.Parent$. Let b be the event such that $b.Node = B$, or `nil` if $B = ROOT$.

- If node A is an **unnamed node**, then:
  - A is a leaf node.
  - B is not *ROOT*.
  - A is one of B's unnamed children.
  - If A is the most recently-created of B's unnamed children, then A is B's rightmost unnamed child.
  - a is in the *Scheduled* state.
  - b is in the *Active* state.
  - a is a direct descendant of b.
- If node A is a **named node**, then **exactly one** of the following is true:
  - B is `nil`, and $A.Parent = ROOT$. Furthermore, there is no indirect rightmost descendant of c of b that is in the *Scheduled* or *Active* state.
  - B is not `nil`, and a is a direct descendant of b. Furthermore, both a and b are in the *Active* state, and A is one of B's named children.
  - B is not `nil`. a is a direct descendant of event c, and b is an indirect rightmost descendant c.

2.5 The `threadsim` Algorithm

The `threadsim` algorithm maintains the invariants specified in section 2.4 by modifying the tree of nodes in response to changes in the states of events.

The algorithm takes as its input an event e, a new state σ_{new} for that event, and a third parameter p that is the event whose handler scheduled e or `nil` otherwise.

The algorithm makes use of the following subroutines:

- $AddRightNamedChild(A, B)$ makes node B the rightmost of node A's named children.
- $AddRightNamedChildren(A, L)$ adds all the nodes in list L to the right of node A's named children.
- $AddRightUnnamedChild(A, B)$ makes node B the rightmost of node A's unnamed children.
- $RemoveUnnamedChild(A, B)$ removes node B from node A's unnamed children.
- $RemoveNamedChild(A, B)$ removes node B from node A's named children.
- $Inherit(A, B)$ takes two nodes A and B, where A is the parent of B and B has no children. This function removes B from A's children and then replaces A with B in the tree of nodes. It then gives names to all the unnamed children of B and adds them to B's named children.

procedure $threadsim(e, \sigma_{new}, p)$

```
1:  σ_old ← e.State
2:  if (σ_old, σ_new) = (Start, Scheduled) then
3:    E ← new Node
4:    e.Node ← E
5:    if p = nil then
6:      E.Name ← new Name
7:      AddRightNamedChild(ROOT, E)
8:    else
9:      E.Name ← nil
10:     AddRightUnnamedChild(p.Node, E)
11:   end if
12: else if (σ_old, σ_new) = (Scheduled, Active) then
13:   E ← e.Node
14:   if E.Name = nil then
15:     E.Name ← new Name
16:     P ← E.Parent
17:     RemoveUnnamedChild(P, E)
18:     AddRightNamedChild(P, E)
19:   end if
20: else if (σ_old, σ_new) = (Active, Canceled) then
21:   E ← e.Node
22:   P ← E.Parent
23:   if E.Name = nil then
24:     RemoveUnnamedChild(P, E)
25:   else
26:     H ← E.RightUnnamedChild
27:     if H = nil then
28:       Name all unnamed children of E
29:       L ← named children of E
30:       AddRightNamedChildren(ROOT, L)
31:     else
32:       Inherit(E, H)
33:     end if
34:   end if
35:   e.Node ← nil
36:   delete E
37: else if (σ_old, σ_new) = (Scheduled, Canceled) then
38:   E ← e.Node
39:   P ← E.Parent
40:   if E.Name = nil then
41:     RemoveUnnamedChild(P, E)
42:   else
43:     RemoveNamedChild(P, E)
44:   end if
45:   e.Node ← nil
46:   delete E
47: else
48:   Error: Illegal state transition.
49: end if
50: e.State ← σ_new
51: Return e
```

2.6 Correctness of `threadsim`

Theorem 2.1 (Correctness of `threadsim`) *The* `threadsim` *algorithm maintains the invariants in section 2.4 across any sequence of legal state transitions for any finite number of events.*

Proof *(By induction on the number of state transitions)*

Let $(e, \sigma_{old}, \sigma_{new})$ denote event e leaving state σ_{old} and entering state σ_{new}.

- *Base Case* If the number of state transitions is 0, then the only node in the tree of simulated threads is $ROOT$, then the invariants are trivially true.
- *Inductive Case* Assume that the invariants still hold after i state transitions. It suffices to show that the invariants hold after $i + 1$ transitions.

  In the cases below, we let E denote $e.Node$ and P denote $e.Node.Parent$. By the inductive hypothesis, both E and P must exist. p denotes the event such that $p.Node = P$, or `nil` if no such event exists.

  - **Case 1:** The $(i + 1)$th transition is $(e, Start, Scheduled)$.
    - **Case 1a:** The event p whose handler is scheduling e is still in the *Active* state. By the inductive hypothesis, there must be a Node in the process

tree that corresponds to p. Lines 3-4 and 9-10 of `threadsim` create a new unnamed node E and make E an unnamed child of P. This new unnamed nodes satisfies all the invariants in section 2.4 that apply to unnamed nodes.

  * **Case 2b:** Event e was scheduled outside of any event handler. Lines 3-4 and 6-7 of `threadsim` create a new named node E that is a child of $ROOT$ and satisfies the invariants that apply to children of $ROOT$.

- **Case 2:** The $(i+1)$th transition is $(e, Scheduled, Active)$.
  * **Case 2a:** E is an unnamed node. By the inductive hypothesis, E must be a leaf node, P must not be $ROOT$, and e must be a direct descendant of p. Lines 15-18 of `threadsim` give E a new name and move E to the named children of P, so the invariants are still satisfied.
  * **Case 2b:** E is a named node. `threadsim` does not modify the process tree, so the invariants are trivially true.
- **Case 3:** The $(i+1)$th transition is $(e, Active, Canceled)$.
  * **Case 3a:** e has a direct rightmost descendant h. Let H denote $h.Node$. By the inductive hypothesis, H must be the rightmost unnamed child of E and must be a leaf node. Line 31 of `threadsim` uses the *Inherit* subroutine to name all of E's children and replace E with H. All of the new children of H are named nodes, and H is an indirect rightmost descendant of their original parents, so the invariants still hold.
  * **Case 3b:** e does not have a rightmost direct descendant. Lines 28-29 of `threadsim` give all of the children of E names and make all those children children of the root. Since there is no longer any indirect rightmost descendant of any of the original parents of these children, making them children of $ROOT$ preserves the invariants.
- **Case 4:** The $(i+1)$th transition is $(e, Scheduled, Canceled)$. By the inductive hypothesis, E must be a leaf node. Lines 37-41 of `threadsim` remove E from the children of P and delete E. Since E has no children, the invariants are still true.

Thus, by induction on the number of state transitions, `threadsim` preserves the invariants in section 2.4 over any valid sequence of state transitions. ∎

Corollary 2.2 `threadsim` *maintains the invariants in section 2.4 regardless of the number of event handlers that execute simultaneously.*

2.7 Running Time of `threadsim`

Theorem 2.3 (Constant overhead per event) *Let n be the number of events that make a transition from the Start state to the Scheduled state. The total running time of* `threadsim` *to process* **all** *the state transitions for these events is $O(n)$.*

Proof For each event e that transitions from *Start* to *Scheduled*, `threadsim` will perform the following actions *at most once*:

- Create a new Node E and set $e.Node \leftarrow E$.
- Assign a fresh name to $e.Node$ and move $e.Node$ from its parent's unnamed children to its parent's named children.
- Call $Inherit(e.Node.Parent, e.Node)$ (See section 2.5). Only an unnamed node can inherit its parent's position, and, in doing so, the node becomes named.
- Make $e.Node$ a child of the root. Once a node is a child of the root, its position will not change.
- Delete $e.Node$, remove $e.Node$ from its parent's children, and set $e.Node \leftarrow$ `nil`.

Each of these operations can clearly be made to run in $O(1)$ time. Thus, `threadsim` adds an overhead of $O(n)$. ∎

2.8 Optimizations

2.8.1 `recycle`

It is common for event-driven systems to execute a sequence of identical events. For example, to receive a large amount of data from a network socket, an event-driven system may use a sequence of *read* events. The `threadsim` algorithm as stated above treats each such *read* event as a distinct entity, creating and destroying a Node for each one. To reduce

the overhead of mapping such repeated events, we instead use the following algorithm to handle them:

procedure *recycle*(e)

1: **if** $e.State \neq Active$ **then**
2: **Error:** *recycle*() only works on events in the *Active* state.
3: **end if**
4: $E \leftarrow e.Node$
5: **for all** F in E's unnamed children **do**
6: $RemoveUnnamedChild(E, F)$
7: $F.Name \leftarrow$ new Name
8: $AddRightNamedChild(E, F)$
9: **end for**
10: $e' \leftarrow$ new Event
11: $e'.State \leftarrow Scheduled$
12: $e'.Node \leftarrow E$ {Recycle node E.}
13: $e.Node \leftarrow$ `nil`
14: $e.Children \leftarrow e.Children + e'$
15: Return e'

Clearly, *recycle*(e) is equivalent to:

1: $e' \leftarrow$ new Event
2: $e.Children \leftarrow e.Children + e'$
3: $threadsim(e', Scheduled)$
4: $threadsim(e, Canceled)$

2.8.2 Eliminating *Children*

The *Children* fields of Events are not necessary to the correct operation of `threadsim`, since the algorithm never reads from those field. The fields can therefore be eliminated.

3 Current Implementation

This section describes the current state of our project. Figure 4 shows the overall architecture of our system.

3.1 The `rlib` Library

We have implemented the `threadsim` algorithm in a library called `rlib`. Our implementation is faithful to the description of `threadsim` given in this paper in most respects. `rlib` also implements the optimizations in section 2.8. Our library is written entirely in C, and we use a memory pool to speed the allocation and deallocation of Nodes.

3.2 Modifications to the Event Loop

`rlib` currently assumes that there is only one event loop. The library exports the following C interface to the event loop:

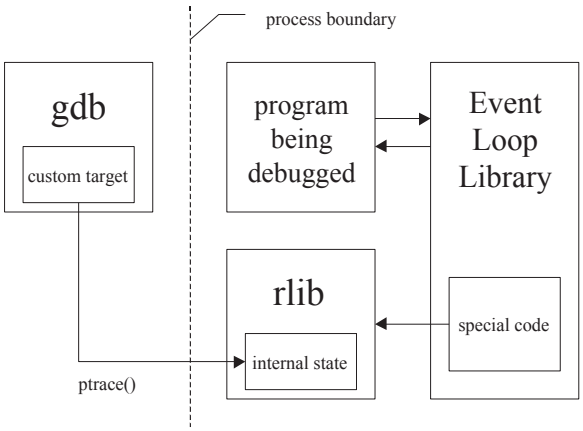

Figure 4: Overall architecture of our project.

- `void rlib_evt_init()` initializes `rlib`.
- `void rlib_evt_sched_event( rlib_uid_t * evt_uid, void * handler)` indicates to `rlib` that an event has entered the *Scheduled* state. `evt_uid` is a event-loop-specific unique identifier for the event. `rlib` stores this identifier in a hash table and uses the table to resolve further references to the event. `handler` points to the beginning of the handler function for the event.
- `void rlib_evt_cancel_event( rlib_uid_t * evt_uid )` indicates to `rlib` that an event has entered the *Canceled* state.
- `void rlib_evt_enter_handler( rlib_uid_t * evt_uid)` tells `rlib` that the indicated event is entering the *Active* state. In order to handle recursive calls into the event loop from handler functions, `rlib` keeps a stack of events that are in the *Active* state. A call to `rlib_evt_enter_handler` causes `rlib` to push a pointer to the indicated event's node onto this stack.
- `void rlib_evt_leave_handler()` indicates to `rlib` that an event handler function has returned. No arguments are necessary, since `rlib` keeps a stack of the events that are in the *Active* state.

We chose to use the event loop of the ACE [12] networking library for our initial implementation. ACE is a collection of C++ wrappers for networking APIs and implementations of common networking design patterns. The library is in active use by several hundred development teams, mostly in the embedded systems community. We inserted calls to the five

functions in `rlib`'s event loop interface into the appropriate parts of ACE, along with approximately 150 lines of code to convert C++ virtual functions into C function pointers and to marshal arguments for the `rlib` API functions.

3.3 Modifications to `gdb`

One of the primary design goals of the `gdb` debugger is portability across machine architectures and operating systems. The debugger acts on the underlying hardware and operating system through an abstraction layer known as the "target" interface. We added a new target to `gdb`, replacing the target that implements Linux thread support with one that simulates threads using information gleaned from the internal state of `rlib`. To track changes to the internal state of the library, our code sets special breakpoints at special "hook" functions inside `rlib`. Implementing this addition to `gdb` proved to be a very difficult task. Some of `gdb`'s assumptions about thread libraries do not hold true for our simulated threads. For example, `gdb` assumes that the frame pointer of a thread will not change unless a `push`, `pop`, or `call` instruction executes. However, the stack pointers of our simulated threads can undergo sudden changes as threads jump from one event handler to another. Remedying this and other "impedance mismatches" required approximately 2500 lines of code.

3.4 Support for Graphical User Interfaces

We have tested our modified version of `gdb` with `ddd` and `kdbg`, two of the most popular and complete graphical front ends for `gdb` and found no incompatibilities. Since our modifications to `gdb` were mostly confined to the low-level, target-specific portions of the debugger, we expect that other graphical front ends for `gdb` will work without modification.

4 Future Work

Our current implementation of debugger support for message-oriented programming is a work in progress. In the short run, we will continue to fix bugs and to add features. Two features we plan to add soon are support for reading core dumps into the debugger and support for event loop libraries other than ACE. We do not anticipate these additions being overly difficult. To date, our project has only been tested under Linux, and we would like to support other operating systems.

The `threadsim` algorithm supports arbitrary overlapping of event handlers, as long as access to the `threadsim` function is serialized. An obvious avenue of future research is to apply `threadsim` towards debugging message-oriented systems with multiple threads of control, especially distributed message-oriented systems like Inktomi [9]. A version of our project for distributed systems would need to simulate thread migration in addition to simulating threads.

Since the operations that `threadsim` performs on Nodes deal almost entirely with immediate parents and children of those Nodes, it should be possible to design a version of `threadsim` for distributed applications that uses more fine-grained locking. Such a locking protocol would allow the algorithm to run on a distributed system with a minimal reduction in concurrency.

5 Related Work

Debugging a message-oriented system using the control flow of process-oriented system is, as far as we know, a new area of research. In this section, we summarize past research that is somewhat similar to the concepts we explore in this paper.

Researchers have developed tools for visualizing streams of messages in message-oriented systems. Some of the visualizations these tools produce are similar to the simulated threads that our project creates. However, these tools are batch systems, unlike `rlib`, which updates the mapping from events to simulated threads in real time. Furthermore, none of these researchers have implemented a debugger that uses their visualizations.

There are several debuggers on the market for message-oriented distributed systems that use remote procedure calls, especially for systems that adhere to the CORBA standard. These debuggers allow the user to step through RPC invocations on remote machines as if the procedures were executing on a local machine. However, stepping through RPC invocations seems to be the full extent of the novel features of these debuggers. A distributed debugger based on the `threadsim` algorithm would be strictly more general.

6 Conclusion

Our project has met its overall goal of allowing programmers to debug a message-oriented system using the more intuitive control flow of a process-oriented system. In addition, the design and implementation

of our project has met the project's more specific design goals:

- Event loops can use the functionality of our project with the addition of as little as ten lines of code. Our `rlib` library handles most of the necessary bookkeeping in a way that is portable across different implementations of the single-threaded event-driven model.
- We have proven that our `threadsim` algorithm works properly for any serial sequence of legal event state transitions. Thus, as long as the `threadsim` function runs in a critical section, the algorithm can handle an arbitrary number of possibly recursive event loops, running in different threads of control.
- We have proven that the amortized overhead `threadsim` is constant time for each event that enters the system. Each event typically results in one or more system calls, and one iteration of `threadsim` should be faster than a system call, so we do not anticipate a significant performance decrease from linking against `rlib` during the debugging phase of a project.
- Since we modified only the low-level, target-specific portions of the `gdb` debugger, our project works with graphical front ends to `gdb` without modification to those front ends.

References

[1] Lauer, H.C., Needham, R.M., "On the Duality of Operating System Structures," in Proc. Second International Symposium on Operating Systems, IRIA, Oct. 1978, reprinted in Operating Systems Review, 13,2 April 1979, pp. 3-19.

[2] Pai, V.S. et al., "Flash: An Efficient and Portable Web Server." USENIX 1999.

[3] Welsh, Matt et al., "A Design Framework for Highly Concurrent Systems." Submitted for publication, April 2000.

[4] Kegel, Dan. "The C10K Problem." `http://www.kegel.com/c10k.html`

[5] Larus, James and Parkes, Michael. "Enhanced Server Performance with StagedServer." Lecture. U.C. Berkeley, October 2000.

[6] Anderson, Thomas E. et al., "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism." ACM Transactions on Computer Systems, 1992.

[7] Kaashoek, M. Frans et al., "Application Performance and Flexibility on Exokernel Systems." 16th Symposium on Operating Systems Principles, 1997.

[8] Bershad, Brian N., et al., "Extensibility, Safety, and Performance in the SPIN Operating System." 15th Symposium on Operating Systems Principles, 1996.

[9] Brewer, Eric. "Lessons from Giant-Scale Services." Submitted for publication, 1999.

[10] Free Software Foundation, `gdb`. Computer program. `http://sources.redhat.com/gdb`.

[11] Schmidt, Douglas C., "Reactor: An Object Behavioral Pattern for Concurrent Event, Demultiplexing and Event Handler Dispatching." August 1994.

[12] Schmidt, Douglas C., "The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software." Sun User Group Conference, 1993.