Debugging Scheme Fair Threads

Damien Ciabrini
INRIA Sophia Antipolis
2004 route des Lucioles - BP 93
F-06902 Sophia Antipolis, Cedex
Damien.Ciabrini@sophia.inria.fr

Abstract

There are two main policies for scheduling thread-based concurrent programs: preemptive scheduling and cooperative scheduling. The former is known to be difficult to debug, because it is usually non-deterministic and can lead to data races or difficult thread synchronization. We believe the latter is a better model when it comes to debugging programs.

In this paper, we discuss the debugging of Scheme Fair Threads, that are based on cooperative scheduling and synchronous reactive programming. In this approach, thread communication and synchronization is achieved by means of special primitives called signals, which ease the debugging process. We present the tools we have implemented to deal with the main types of concurrent bugs that can arise in this special programming framework.

1 Introduction

Modern systems offer multitasking inside a single application: there can be many virtually independent flows of control, usually called *threads*. These are commonly used in programs nowadays.

Concurrent programming is a difficult task. First, because reasoning about interleaved flows of control is an intrinsically difficult task. Second, because bugs caused by multi-threaded programming are usually very difficult to track down with traditional debuggers.

There are various policies for scheduling multi-threaded programs. The two major categories are *preemptive* scheduling and *cooperative* scheduling. Each one comes with its pros and cons with respect to debugging.

1.1 Preemptive or Cooperative Scheduling

Preemptive scheduling appeared in operating systems [13] in the late 70s and has been democratized in languages in the mid 90s. In this model, the thread library (usually the underlying OS) may suspend the execution of a thread at any time to schedule another one. It can also benefit from Symmetric Multi-Processor hardware (henceforth SMP). Unfortunately, preemptive multi-threading is implemented in a way that leads to non-deterministic scheduling. It is known to be difficult to program with and painful to debug:

- Locks have to be acquired before accessing shared memory to avoid data races. Omitting locks may cause data corruption, in which case debuggers become almost useless.
- Synchronization by means of *mutexes* can be missed if notifications are sent before some threads started to await them. In this case, debuggers hardly help because they do not provide tools for tracing the order of synchronization.
- It is very difficult to reproduce a bug because one cannot play the same execution twice. Actually, the simple fact of inserting `prints` in a program is sufficient to make a bug no longer appear at run-time.
- Complex features like priority boost or scheduling policies are non-portable, and debuggers usually do not provide support for them. Using these features can lead to bugs like priority inversion [20], that are difficult to track or to explain.

Cooperative scheduling is an older model, in which it is the responsibility of threads themselves to *cooperate*, *i.e*, to give back control so that another thread can continue to execute. This is a deterministic model where only one thread is executing at a time (which hardly benefits from SMP). This scheduling model greatly eases the debugging for various reasons:

- Debuggers do not have to deal with data races, since only one thread is active at a time.
- The scheduler is deterministic. This means that when a bug occured, it can be easily reproduced by replaying the same execution.

In cooperative schedulers, problems like dead-locks can still occur. Moreover, some specific problems are introduced because of manual cooperation:

- If a thread fails to cooperate, the whole program is blocked.
- Too few cooperations can lead to interactivity problems, for instance in Graphical User Interfaces (henceforth GUI).
- Too many cooperations can lead to unnecessary context switches and poor performance. This is a problem similar as taking too many locks to protect shared variables.

Contrary to bugs caused by non-determinism, these types of bug are much easier to detect and to correct with the help of a debugger.

Permission to make digital or hard copies, to republish, to post on servers or to redistribute to lists all or part of this work is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To otherwise copy or redistribute requires prior specific permission.
Fifth Workshop on Scheme and Functional Programming. September 22, 2004, Snowbird, Utah, USA. Copyright 2004 Damien Ciabrini.

1.2 An Hybrid Solution

The Bigloo [18] Scheme compiler provides an alternative model for multi-threaded programming called Scheme Fair Threads [17]. It is a programming framework based on *synchronous reactive programming*[1] with the following characteristics:

- It provides a cooperative and deterministic scheduler. Thus there is no need to acquire locks and execution can be replayed at will.
- Threads communicate by broadcasting *signals* into the scheduler. It is guaranteed that signals are seen by all threads during a logical round of schedule called an *instant*.
- It still provides the ability to do I/O operations asynchronously, *i.e*, without blocking the scheduler and also by taking advantage of SMP. Non-determinism due to asynchronous I/Os is confined into well defined locations in the scheduler.

In Scheme Fair Threads, a debugger still has to deal with deadlocks or live-locks, as it is the case with classical cooperative or preemptive scheduling. However, it does handle communication and synchronization bugs differently, because mutexes are abandoned in favor of broadcast signals. The debugger has to provide a new set of tools to deal with specific problems introduced by this programming framework.

We have extended BUGLOO [2], a source-level debugger for Bigloo programs, in order to support the debugging of fair threads. In this paper, we do not talk about POSIX-like mutexes or condition variables, as Fair Threads do not use them to communicate. Instead, we concentrate on the debugging of cooperation points, signals and instants:

- We have enhanced the single stepping by introducing new step points that take into account cooperative scheduling and communication by broadcast signals.
- We provide a tool for inspecting the state of fair threads and for viewing signals present in the scheduler when execution is suspended. This tool is used to fix bugs like dead-locks and live-locks that occur *during* a single instant.
- We provide a tool to graphically trace the scheduling of fair threads and the broadcasting of signals throughout the execution. It is an effective way to fix communication bugs that occur *across* many instants.

1.3 Overview

In Section 2, we detail the Fair Threads programming model and its usage in Scheme. In Section 3, we present the main debugging support for Fair Threads, namely the single stepping and state inspection. In Section 4, we describe the tool for tracing scheduler executions. In Section 5 we describe the typical usage of our tools on a producer-consumer program with asynchronous I/Os. In Section 6 we briefly describe how the debugger is implemented and we present the overall experience we had with our tools. In Section 7 we present related work. Finally, Section 8 concludes and shows some future directions for our work.

[1] See http://www-sop.inria.fr/mimosa/rp

2 Scheme Fair Threads

Scheme Fair Threads is a thread-based concurrent programming framework based on Java Fair Threads [1]. In this section we present the Fair Threads programming framework and the concept of fair scheduling. As an example, we describe the execution of an abstract program. We then give a brief overview of the Fair Threads API, along with the type of concurrent bugs that can occur in this programming framework.

2.1 Fair Threads and Fair Scheduling

In the Fair Threads model, each fair thread is mapped to a native OS thread and has its own dynamic environment. Threads are *attached* to a cooperative scheduler, in which only one thread is executed at a time. When a thread cooperates, the scheduler gives control to another thread. Note that the scheduler is deterministic, and is itself a fair thread.

The Fair Threads model has a clear semantics that emphasizes *fair* scheduling and powerful means of synchronization. Both principles are described below:

- The scheduling of fair threads is decomposed into logical units of schedule called *instants*. During an instant, fair threads communicate together by awaiting and broadcasting *signals* into the scheduler. A signal is present until the end of instant and can be associated with a value.
- If a signal is broadcast during an instant, all the threads waiting for it are guaranteed to be notified before the end of the instant. In particular, if a fair thread waits for a signal that has already been broadcast in the instant, it is immediately notified and continues its execution.
- A signal can be broadcast many times in the same instant. Fair threads can wait until the next instant to obtain the list of all the values associated with a signal that were generated in the previous instant.
- A fair thread can be re-elected for schedule in the same instant if signals have been broadcast since its last election. An instant terminates when all the fair threads have been executed and no new signal has been broadcast.

2.2 A Simple Program

To understand how fair threads are scheduled, let us describe the execution of the following abstract program composed of three fair threads.

```
        A                B                C
--------------   -----------------   ---------------
1: await sig1    1: broadcast sig1   1: await sig1
2: await sig2    2: yield            2: broadcast sig2
3: yield         3: broadcast sig3   3: await sig3
4: await sig1
```

Instant 1: fair thread *A* gets blocked in line 1 waiting for signal `sig1` to be broadcast. Next, fair thread *B* broadcasts `sig1` in line 1, then it cooperates in line 2 to explicitly complete its execution for the instant. At this point, the scheduler re-elects fair thread *A* because signal `sig1` has been broadcast in the scheduler. This fair thread then blocks waiting for signal `sig2`. Then, fair thread *C* takes the control. It does not block on signal `sig1` because it has already been broadcast (by *B*) during the instant. It broadcasts `sig2` and blocks, waiting for signal `sig3`. At this point, the instant is not

over: the scheduler re-elects fair thread *A* because signal `sig2` is now present. Then, this fair thread explicitly cooperates in line 3. At this very point all threads are blocked and no new signal has been broadcast. This marks the end of instant 1. *Signals are reset*.

Instant 2: fair thread *A* gets blocked in line 4 waiting for signal `sig1` which has not been broadcast yet during the instant. Fair thread *B* broadcasts signal `sig3` then it terminates its execution. Fair thread *C* is awakened by signal `sig3` in line 3 and it terminates. At this point all threads are blocked, the instant *2* ends and signals are reset.

Instant 3: fair thread *A* is still blocked for the instant and for the remaining instants until somebody broadcasts `sig1`.

2.3 API Overview

The Fair Threads API has been designed to be fully compatible with the SRFI-18 by M. Feeley [5]. This document proposes an extension for multi-threaded programming in Scheme, inspired by the Posix-1 API and the Java API. In Fair Threads, abstraction like mutexes or condition variable are implemented on top of signals. The previous abstract example is implemented in Fair Threads as followed:

```
 1: (define (funA)
 2:    (thread-await! 'sig1)
 3:    (thread-await! 'sig2)
 4:    (thread-yield!)
 5:    (thread-await! 'sig1))
 6:
 7: (define (funB)
 8:    (broadcast! 'sig1)
 9:    (thread-yield!)
10:    (broadcast! 'sig3))
11:
12: (define (funC)
13:    (thread-await! 'sig1)
14:    (broadcast! 'sig2)
15:    (thread-await! 'sig3))
16:
17: (define (main args)
18:    (thread-start!
19:       (make-thread funA "fairthread A"))
20:    (thread-start!
21:       (make-thread funB "fairthread B"))
22:    (thread-start!
23:       (make-thread funC "fairthread C"))
24:    (scheduler-start!))
```

We now describe the major constructions of the Fair Thread API.

2.3.1 Basic Thread Manipulation

As shown in the previous example in line 19, a fair thread is created with the `(make-thread` *`thunk`* `.` *`name`*`)` procedure, which takes a *`thunk`* to execute and an optional *`name`*. A fair thread must be started with `(thread-start!` *`thread`*`)` before it can be executed by the scheduler.

Cooperation is achieved by calling the `(thread-yield!)` procedure, as shown in lines 4 and 9. One thread can terminate another thread with the `(thread-terminate!` *`thread`*`)` procedure.

Unlike many threading systems, the scheduler has to be started explicitly with `(scheduler-start!)`. When started, the scheduler runs until all its threads are completed or terminated.

2.3.2 Communication by Signals

A fair thread can broadcast a signal into the scheduler with the `(broadcast!` *`sig`* `.` *`value`*`)` procedure. A signal can be an arbitrary Scheme object. The broadcast can be associated with an optional *`value`* that will be received by waiting threads on awake. The default value is the symbol `#unspecified`, to indicate that no particular value is associated with the broadcast of the signal.

A fair thread can await a signal by means of the `(thread-await!` *`sig`*`)` procedure. It can also await several signals at a time with `(thread-await!*` *`sigs`*`)`. At last, a fair thread can get all the values broadcast in the instant for a particular signal by using the `(thread-get-values` *`sig`*`)` procedure. The fair thread waits until the end of the current instant, and at the next instant it is awakened with the list of broadcast values.

Mutexes and condition variables are implemented on top of signals and are not presented in detail in this paper. They are still accessible by their respective SRFI-18 procedures.

2.3.3 Asynchronous I/O and SMP

Fair threads can start special *service threads* whose purpose is to do long lasting I/O operations in the background without blocking the scheduler. Such threads are standard OS threads that benefit from SMP. No lock is needed in user space because service threads cannot execute user procedures.

On I/O termination, a signal is broadcast into the scheduler to awake the fair thread that requested the operation. Here is a subset of the service threads currently supported:

- *output*: `(make-output-signal` *`p s`*`)` spawns a service thread that writes the string *s* to the output port *p*.
- *input*: `(make-input-signal` *`p n`*`)` spawns a service thread that gets *n* characters from the input port *p*.
- *socket*: `(make-accept-signal` *`s`*`)` spawns a service thread that waits for a connection on the socket *s*.
- *process*: `(make-process-signal` *`p`*`)` spawns a service thread that forks process *p* in the background.

By definition, using asynchronous I/Os introduces a certain kind of non-determinism. However, it is not harmful because it is confined into service threads, thus it cannot cause any data corruption in user space. Moreover, the fairness of the scheduler is maintained since from a thread's point of view, being notified of an I/O termination is exactly the same thing as being awakened by a signal.

2.4 Classification of Fair Threads Bugs

We saw that with Fair Threads, communication or synchronization is always based on signals and instants instead of mutexes and condition variables. In this framework, the type of bugs that can be caused by multi-threading can be classified in two subsets:

1. Bugs that can be fixed by inspecting the state of the program in the *current instant*. An example of such bug could be a deadlock that occurs because all fair threads are awaiting signals.

It could also be a bug caused by a fair thread which is stuck in a live-lock, *i.e*, a thread that repeatedly waits for a signal that has already been broadcast, thus preventing the instant to terminate. To fix this kind of errors, we provide two tools: an enhanced single stepper and a scheduler and fair thread inspector. Both are presented in Section 3.

2. Bugs for which one needs to remember the state of the program *several instants backward* in time. For example, in a badly designed sequence of successive communications, a fair thread can await a signal in a particular instant while it was broadcast in a previous instant. Dead-locks and live-locks can also be caused by a succession of wrong synchronization. To fix this kind of errors, we provide a tool to graphically visualize what happened in the scheduler during a succession of instants. It is presented in Section 4.

3 Debugging Fair Threads

In this work, we have included debugging support for Fair Threads into BUGLOO, a debugger for Scheme programs compiled into Java VM [12] bytecode. BUGLOO is a complete source-level debugger with a command line language. It is integrated in the Bee development environment [16], and is meant to be used from Emacs or Xemacs.

The tools we have implemented are displayed in a new GUI layer which is used in conjunction with Emacs. It is implemented in Biglook [7].

In this section, we show how to start a debugging session and we present the first two debugging tools we have implemented: an enhanced single stepper and a scheduler and fair thread inspector. They can be used to fix bugs like dead-locks or live-locks that may occur during an instant. In the followings of this paper, the term *debuggee* will denote the program that is being debugged.

3.1 The Fair Threads Debugging Toolbox

A typical debugging session consists in connecting an Emacs buffer with BUGLOO, setting breakpoints somewhere in the source and running the program. Let us suppose that we ran the little program presented in Section 2.3 and that the execution was suspended on a breakpoint line 13. Then, the user can pop up the Fair Threads toolbox showed in Figure 1, from which all the debugging tools are accessible. From top to bottom, the toolbox contains a set of buttons for enhanced single stepping, buttons to display traces of scheduler executions, and a list of fair threads present in the program. We will now describe these tools.

Figure 1. The *Fair Threads* debugging toolbox

3.1.1 Enhanced Single Stepping

Signals and instants introduce new logical points of control in the execution. We have thus enhanced the classic single stepping operation by providing six new possible step points accessible through buttons in the toolbox:

- **End of Instant** continues the execution until the end of the current instant, and suspends the debuggee just before the next instant begins. It is useful to see the state of fair threads or all the broadcast signals at the end of an instant;
- **Beginning of Instant** suspends the execution as soon as a new instant is started. It allows one to quickly step up to a point that will be single stepped more precisely for debug purposes;
- **Thread Switch** suspends the execution as soon as a new fair thread gets the control. It is useful to see how threads are scheduled during an instant;
- **Next Await** continues the execution until any thread awaits a signal. It can be used to single step a communication mechanism;
- **Next Broadcast** continues the execution until a thread broadcasts a signal in the scheduler. It is the dual of the previous step action;
- **Any Scheduler Event** suspends the execution on any of the preceding event.

3.1.2 Trace of Events

Throughout the execution, scheduler events like thread switch, signal await, signal broadcast or end of instant can be recorded. Our interest in tracing these events is twofold:

1. It enhances the debugging information provided by the fair thread inspector that will be presented in Section 3.2. For instance, it allows the debugger to remember which thread is responsible for a particular broadcast, along with its location in the source at this time.
2. It allows one to understand what happened precisely in the scheduler across *many* instants, and to analyze this information *off-line*.

In the debugging toolbox, two options can be checked to control the recording of events.

- **Enhanced inspection**. When checked, the recording of events is activated as soon as the execution is suspended. When the user starts single stepping the program, he automatically gets enhanced information in the inspectors[2]. Enhanced inspection is automatically switched off as soon as the execution is resumed, to avoid performance penalties during normal execution. In this trace mode, recorded events are reset every new instant.
- **Record Scheduler State**. When checked, the event recording stays activated during execution and across instant boundaries. Later, the resulting trace can be cleared or shown by clicking on the appropriate buttons (see Figure 1).

3.1.3 List of Fair Threads

The last part of the debugging toolbox shows the lists of live fair threads (this frame does not show native OS threads present in the program). The list is arranged into a tree where the directory nodes represent the schedulers, and the leaves represent the attached fair threads. Note that there may be several schedulers in a program, and that schedulers can be nested, since they are actually specialized fair threads.

Figure 1 shows the three fair threads present in the previous program, plus their scheduler. Threads are identified by their name, or by a unique thread descriptor that can be used in the BUGLOO command line. Double-clicking on a node opens a *fair thread inspector* in a new window. It is described below.

3.2 Fair Thread Inspector

An inspector provides a graphical representation (henceforth called a *view*) of the state of a debuggee object at the time the execution was suspended. BUGLOO provides various specialized views for different object types. In particular, we have implemented views for the three main types introduced by Fair Threads: schedulers, fair threads and signals. Below, we describe the basic services provided by an object inspector. Then we present the specific Fair Threads views.

3.2.1 Object Inspectors

Inspectors are top-level windows that provide a set of common features and attributes. Figures 2, 3 and 4 show screenshots of different inspectors.

The bottom status bar shows the type of the inspected object. In a view, fields that point to other debuggee objects are themselves inspectable. A common pop-up menu lets the user inspect objects within the current inspector window or in a new one, as show in Figure 2. The toolbar at the top of the view provides a set of generic actions available in every inspector:

- When the user inspects a new object in the same inspector window, the old view is kept in a view history and is accessible through the top toolbar. The history is managed in a browser-like fashion: one can go backward or forward. When a new view is created, it is inserted at the current point in history, and all the views forward this point are forgotten.
- A particular object type can be associated with many graphical views. For instance, a scheduler can be visualized as a simple fair thread or as a scheduler (more specific view). The third button in the toolbox can be used to change the current view (see Figure 3). A default view is provided for all types. It is basically an object inspector and is not presented in this paper.

[2]Enhanced debugging is only fully effective at the beginning of the next instant.

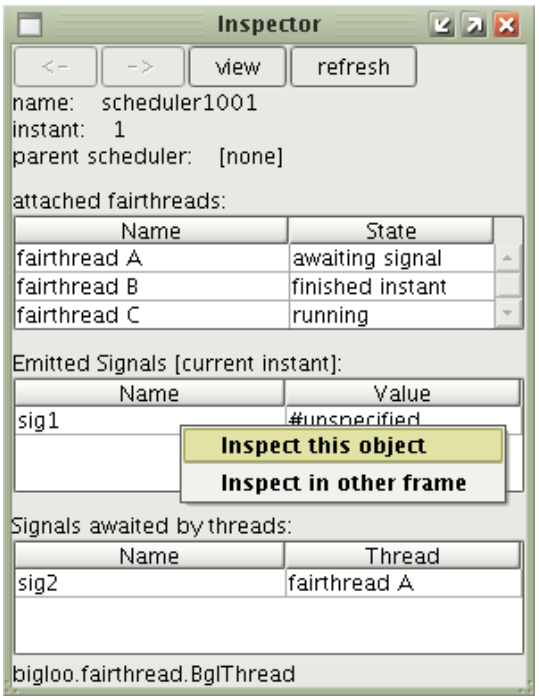

Figure 2. Scheduler Inspector

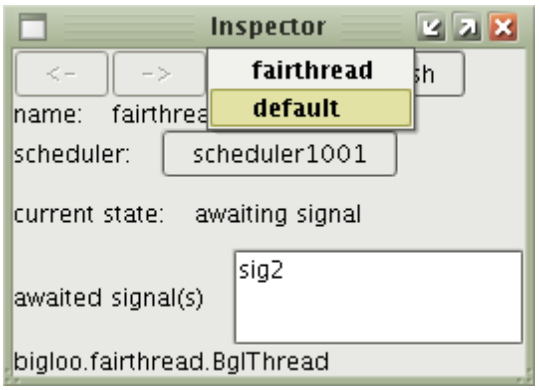

Figure 3. Fair Thread Inspector

3.2.2 The Scheduler View

The screenshot shown in Figure 2 represents the state of the scheduler when the program presented in Section 2.3 is suspended at line 13. The scheduler view is decomposed in four parts. The first part exposes basic information concerning the scheduler: its name, the current instant at the time the execution was suspended, and its par-

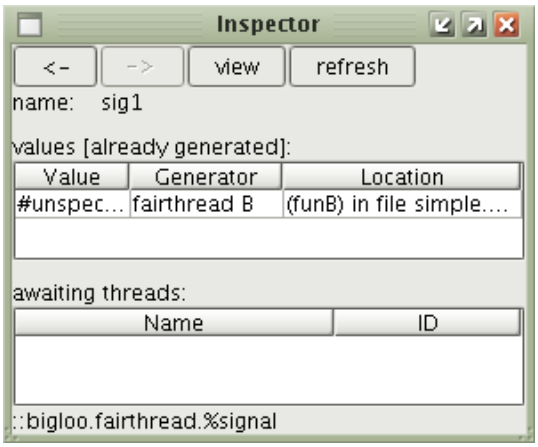

Figure 4. Signal Inspector

ent scheduler, if any (or the symbol `[none]`).

The second part is a table widget that shows all the fair threads attached to this scheduler. Double-clicking on a line pushes a new view of the selected fair thread in the inspector. Information about a fair thread includes its name and its current state in the scheduler. Unlike POSIX threads, a fair thread can be in six different states:

- **running**: the fair thread is currently executing;
- **standby**: the fair thread is eligible for execution during the instant;
- **await**: the fair thread is awaiting signal(s);
- **end of instant**: the fair thread has terminated its execution for the current instant;
- **terminated**: the fair thread has terminated its entire execution;
- **unattached**: the fair thread has not been started yet, because it is not attached to a scheduler. Obviously this state can only be seen in the fair thread view presented further on.

The last two parts of the inspector are devoted to signals.

- A first table represents signals that have been broadcast in the scheduler during the instant. Information about a signal includes its name and its value (or `[...]` if the signal has been broadcast several times). Double-clicking on a line pushes a new view of the selected signal in the inspector.
- A second table represents signals that are awaited by threads, and that have not been broadcast in the scheduler yet. As soon as an awaited signal is broadcast, its entry in the table migrates to the first table. An entry is composed of the signal's name and its awaiting threads.

3.2.3 The Fair Thread View

The fair thread view presented in Figure 3 is quite simple. It first shows the name of the fair thread, and that of its scheduler. If the latter is clicked, a new view is pushed on the inspector. The view also shows the state of the fair thread. It can be any of those presented in the scheduler view. Moreover, if the thread is awaiting one or more signals, their names are displayed in a list-box.

In the screenshot, we see that fair thread `A` is awaiting signal `sig2`. Using many inspectors at a time, one can visualize in detail the state of several fair threads.

3.2.4 The Signal View

The signal view is composed of the name of the inspected signal and of two other tables. The first table contains the different values associated with each broadcast of the signal in the current instant. If enhanced inspection is enabled, each signal broadcast comes with additional information: the fair thread that broadcast the signal and its location in the source at the time of the broadcast. In the screenshot of Figure 4, we see that fair thread `B` has broadcast signal `sig1` from function `funB`.

The second part of the inspector lists the threads waiting for this signal. If the signal has already been broadcast in the instant, the table is empty.

4 Tracing the Scheduling of Fair Threads

We already stated that the Fair Threads framework provides stronger means of synchronization that mutexes, because during an instant broadcast signals are seen by all threads.

In Section 3, we presented a set of tools to address communication or synchronization problems that can occur during a *single* instant. However, the user might need to remember what has occurred several instants backward in time to understand the cause of a particular bug. These tool are not designed to provide such information.

In this section, we present a trace tool that is an effective way to visualize the state of a scheduler *inside* and *between* instants. It gives the user a sharp vision of both the scheduling and the communication between fair threads throughout the execution.

4.1 The Trace Tool

When the trace tool is enabled in the debugging toolbox (Figure 1), the user can display a graphical view of a scheduler execution.

For the sake of the example, let us run the little program presented in Section 2.3 and trace its whole execution. We previously stated that this program never terminates because one thread is waiting for a signal while the others have already terminated.

In presence of a dead-lock, the typical action is to force the suspension of the execution by hitting `CTRL+C`, and then requiring BUGLOO to display the recorded trace. The result is shown in Figure 5. The trace is displayed as a graph:

- The vertical axis shows the fair threads attached to a scheduler and all the signals that were broadcast during the recorded execution slice. Signals always appear at the top, followed by the scheduler[3] and the fair threads.
- The horizontal axis represents the progression of the execution across the instants. Instant boundaries are delimited by thick vertical grey lines, along with their respective number at the bottom.

In the trace view, the execution is decomposed into logical units

[3]The scheduler appears in the trace because it is itself a fair thread.

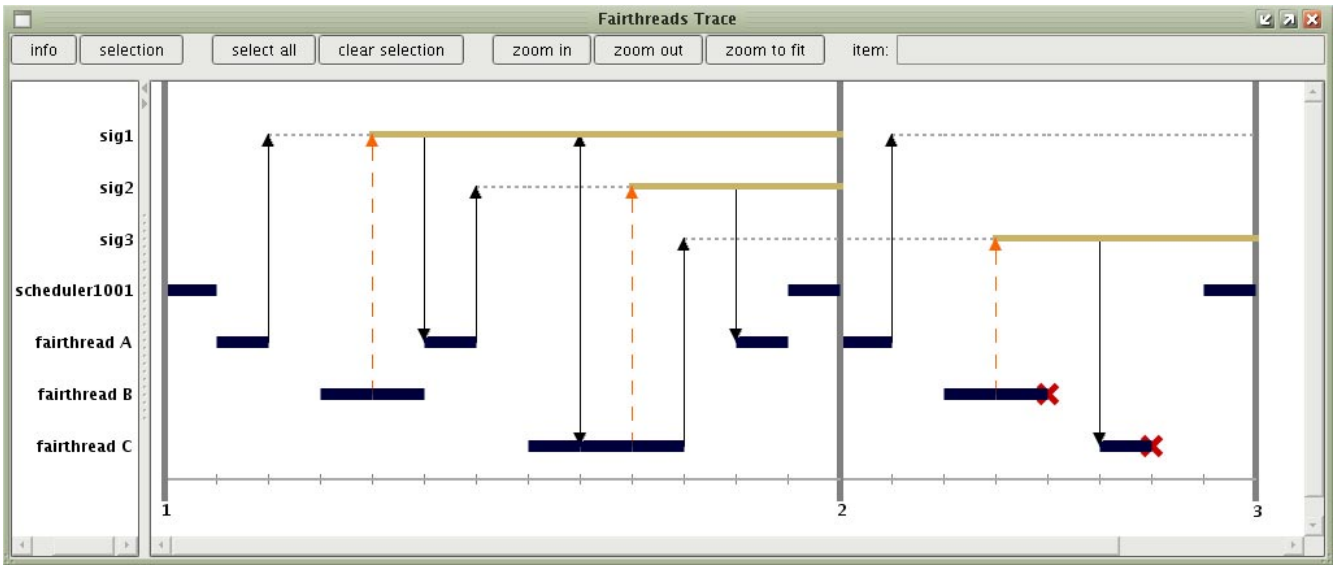

Figure 5. The Trace of the Example shown in Section 2.3

that represent atomic operations that occur inside a scheduler. For example, such units can denote a context switch, the broadcast of a signal, the waiting for a signal, the termination of a thread or the start of an asynchronous I/O operation.

We now explain how to interpret the trace while we describe the important parts of the execution:

At the beginning of instant 1, the scheduler named `scheduler1001` has the control of the execution. The control is symbolized by a thick black horizontal segment. Then, the scheduler allocates the processor to fair thread `A`. This is symbolized by another black segment.

Next, fair thread `A` awaits signal `sig1`, which suspends its execution. The waiting is symbolized by a black vertical arrow that points to the life line of signal `sig1`. A dotted horizontal line is drawn to indicate that this signal has not been broadcast yet during the instant.

Next, the control switches to fair thread `B` which broadcasts signal `sig1` into the scheduler. This is symbolized by a dashed arrow pointing to the life line of `sig1`. To mark the presence of the signal, a thick horizontal line is drawn up to the end of the instant. Remember that broadcasting a signal does not suspend a fair thread. Thus, fair thread `B` has to cooperate explicitly. Overall, it has executed 2 logical operations in the scheduler, hence the double size of the black segment.

Now that signal `sig1` has been broadcast, fair thread `A` is re-elected and continues its execution. The awaking is symbolized by a vertical line starting from the signal life line and pointing to fair thread `A`.

Later in the trace, the control switches to fair thread `C`, which then awaits signal `sig1`. As this signal is already present in the scheduler, the fair thread can continue its execution in sequence. This phenomenon is symbolized in the graph by a double-headed vertical arrow.

When no other fair thread can be scheduled, the scheduler takes back the control and the instant is over.

At the beginning of instant 2, all the signal are reset, thus their respective life lines are empty. Fair thread `A` receives the control and awaits signal `sig1`.

At the bottom right of the graph, one can distinguish two little diagonal crosses at the end of the schedule of fair threads `B` and `C`. This indicates that both threads have terminated their entire execution and will no longer be scheduled.

There is no instant 3. Indeed, the only remaining fair thread in the scheduler is fair thread `A`. Unfortunately, this thread cannot be scheduled because it is awaiting a signal that will never be broadcast from now on. This leads to a dead-lock.

5 Bugloo in Action

In this section we present a complete debugging session on the classical producer-consumer problem. Several producers write data into a global shared buffer of unlimited capacity. Several consumers can read this data and print it using asynchronous I/O operations. We split the complexity of the problem by presenting successively refined implementations, along with the typical synchronization bugs that may arise during this process and how we can track them down with our tools.

5.1 First Implementation

First of all, let us model the problem in Fair Threads. The shared buffer is a simple Scheme list. For the moment, we consider that I/O operations are synchronous. Producers and consumers are naturally modeled as fair threads that put (resp. get) data into (resp. from) the buffer and then cooperate:

```
 1: (define (buffer-fetch)
 2:    (let ((r (car *buffer*)))
 3:       (set! *buffer* (cdr *buffer*))
 4:       r))
 5:
 6: (define (buffer-put! val)
 7:    (if (null? *buffer*)
 8:        (set! *buffer* (list val))
 9:        (set-cdr! (last-pair *buffer*)
10:                  (list val))))
11:

12: (define (make-producer count name)
13:    (make-thread (lambda ()
14:                     (let loop ((n count))
15:                        (put n)
16:                        (thread-yield!)
17:                        (loop (+ 1 n))))
18:                 name))
19:
20: (define (make-consumer name)
21:    (make-thread
22:     (lambda ()
23:        (let loop ()
24:           (print (current-thread) ": " (get))
25:           (thread-yield!)
26:           (loop)))
27:     name))
```

The notification mechanism will occur by the means of the two procedure calls `(put n)` and `(get)`. In the first implementation, the communication model follows a simple wait/notify scheme: on data availability, a signal is broadcast to awake all the consumers.

```
28: (define (wait sig)
29:    (thread-await! sig))
30:
31: (define (notify sig)
32:    (broadcast! sig))
33:
34: (define (put val)
35:    (buffer-put! val)
36:    (notify 'available))
37:
38: (define (get)
39:    (if (buffer-empty?)
40:        (begin
41:           (wait 'available)
42:           (thread-yield!)
43:           (get))
44:        (buffer-fetch)))
```

Note that the `thread-yield!` line 42 is mandatory after the `wait`. Indeed, when a consumer awakes, data may have been already consumed by another consumer. If there was no cooperation, the consumer would retry another `wait` in the same instant. Because a broadcast signal is present until the end of instant, the consumer would not block anymore on signal `available` and would cause a live-lock.

The following piece of program creates producers and consumers and starts the scheduling:

```
(define (start)
   (thread-start! (make-consumer "cons1"))
   (thread-start! (make-consumer "cons2"))
   (thread-start! (make-consumer "cons3"))
   (thread-start! (make-consumer "cons4"))
   (thread-start! (make-producer 1000 "prod1"))
   (thread-start! (make-producer 0 "prod2"))
   (scheduler-start!))
```

When we run the program, the following output is printed on the screen:

```
#<thread:cons4>: 1000
#<thread:cons3>: 0
#<thread:cons2>: 1
#<thread:cons1>: 1001
#<thread:cons2>: 1002
#<thread:cons4>: 2
#<thread:cons3>: 3
#<thread:cons1>: 1003
#<thread:cons2>: 1004
#<thread:cons4>: 4
#<thread:cons3>: 5
```

The first reaction when seeing this output is to think that something went wrong in the scheduling of the producers. Actually, one might assume that producers generated two values in a single instant.

The trace shown in Figure 6 helps to find out why the threads are interleaved this way. It turns out that producers broadcast their signals correctly. In fact, the trace reveals that from an instant to another, fair threads are scheduled in the exact opposite order, which gives the impression of erroneous executions.

In conclusion, we showed that the trace tool is useful to understand the interleaving of threads inside the scheduler. It showed that one should not assume any particular execution order within an instant: the scheduler is deterministic in the sense that another execution will lead to the very same interleaving of fair threads.

5.2 Improving Notification

So far, data availability is signaled to all fair threads. This leads to unnecessary context switches. We can improve the mechanism by putting consumers in a queue, and by signaling availability only to the first thread in this queue. We thus modify the former code as follows:

```
(define (queue-empty?)
   (null? *queue*))

(define (queue-push! val)
   (set! *queue* (append! *queue* (list val))))

(define (queue-pop!)
   (let ((th (car *queue*)))
      (set! *queue* (cdr *queue*))
      th))
```

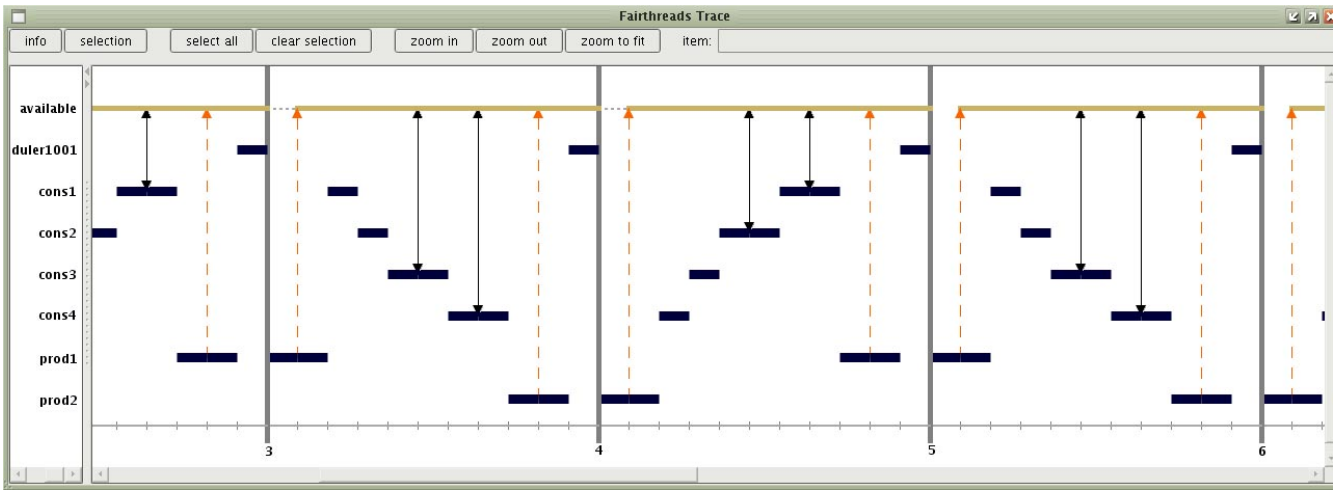

Figure 6. The Trace of the Producers-Consumers with a Naive Implementation.

```
(define (wait sig)
   (let ((self (current-thread)))
      (queue-push! self)
      (thread-await! self)))

(define (notify sig)
   (if (not (queue-empty? sig))
       (broadcast! (queue-pop! sig))))
```

Since any Scheme value can be used to denote signals, we can make each fair thread waiting for a different signal, which is its own thread descriptor returned by `current-thread`. This way, a broadcast signal only awakes one consumer at a time.

5.3 Introducing Non-Determinism

We now replace the consumer's `print` statement at line 24 with a `make-output-signal` to support asynchronous I/O (see Section 2.3.3). This way, the output operation is executed in parallel (*i.e*, preemptively) and does not block other running fair threads, for instance in case the output port is a very slow socket.

We also decide to remove the `thread-yield!` statement line 25, as a call to `make-output-signal` always does an implicit cooperation. The consumer code is rewritten as follows:

```
(define (make-consumer name)
   (make-thread
    (lambda ()
       (let loop ()
          (thread-await!
           (make-output-signal
            (current-output-port)
            (concat (current-thread) ": " (get))))
          (loop)))
    name))
```

When we run the program, the following output is printed on the screen:

```
#<thread:cons4>: 0
#<thread:cons3>: 1000
#<thread:cons3>: 2
#<thread:cons2>: 1002
#<thread:cons4>: 1001
#<thread:cons1>: 1
#<thread:cons1>: 1003
#<thread:cons2>: 3
#<thread:cons2>: 1004
#<thread:cons2>: 5
#<thread:cons3>: 4
```

The output seems coherent. In particular, the new interleaving order and the inversion of data `1002` and `1001` can be explained by the introduction of asynchronous I/O operations. Actually, this modified program contains a subtle bug which will be explained in detail in the following section.

5.4 Profiling the Scheduling

At this time, the program seems bug-free, but the trace tool is still useful to do some profiling analysis. Figure 7 exposes the behavior of the scheduler when running the new program. By observing the trace, we can draw various conclusions:

- The new notification mechanism is working as expected: each producer awakes only one consumer by broadcasting a specific signal.
- Asynchronous I/O operations are materialized in the trace at instant 2 by little diagonal arrows drawn at the extremities of execution segments. An outgoing arrow means that a fair thread started an asynchronous I/O operation and is awaiting its termination. This operation always implies an implicit cooperation. An ingoing arrow means that the consumer can continue its execution because the I/O terminated.
- The trace reveals that asynchronous I/Os can terminate in the instant they were started. This is problematic, because this allowed consumers 1 and 2 to react many times in the same instant and thus to consume more data than they were intended to. We conclude that it was wrong to remove the call to `thread-yield!` line 25 in the consumer code (Section 5.1).
- The trace also reveals that as soon as a consumer is awak-

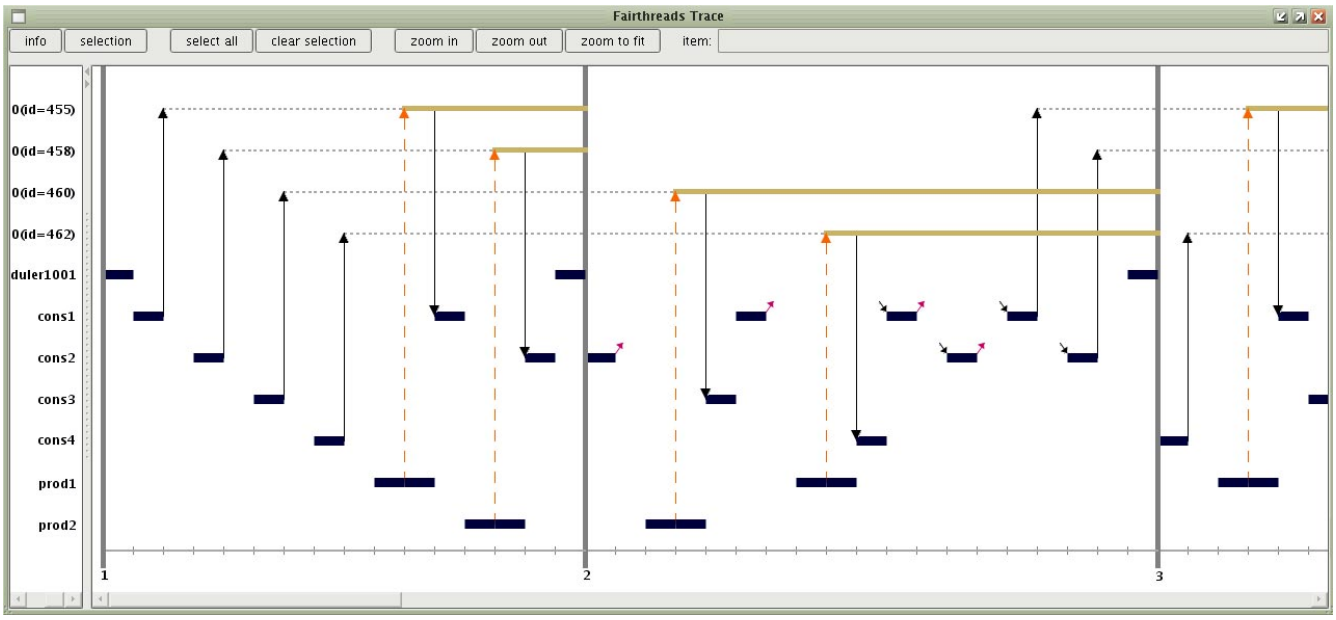

Figure 7. The Trace of the Producers-Consumers with Enhanced Notification and Asynchronous I/Os.

ened, it explicitly cooperates (see `cons2` near the end of instant 1). Then, when it takes back control, it does an asynchronous I/O. The trace tells us that the cooperation is due to the `thread-yield!` located in the `get` function (line 42). In Section 5.1, we stated that this cooperation point was mandatory because a consumer could be awakened while data was no longer available. With the new notification mechanism, this is no longer true. We conclude that this `thread-yield!` leads to superfluous context switches.

In conclusion, we believe that the trace inspector is an effective way to debug communication between threads during execution. It can be used to track down algorithmic bugs, and it can also be employed as a simple profiling tool, since it helps tweaking the cooperation points in a program.

6 Practical Experience

In this section, we briefly explain how the debugger is implemented. We describe the JVM debugging architecture, and how we hook BUGLOO in the debuggee to debug fair threads. In a second part, we present the practical experience we had with our tools and their current limitations.

6.1 Implementation

The debugging architecture of the JVM is close to the one found in GDB [21]: the debugger runs on a JVM, and it instruments the execution of the debuggee which runs in a second JVM. This allows BUGLOO to stay as unintrusive as possible. Debuggee's events like breakpoint hits, single steps or method entries are transmitted to the debugger through an event queue connected to both JVMs. In order to use BUGLOO, it is not necessary to compile the source program in a special debug mode, nor to operate source-code instrumentation on it. The debugger queries remotely the debuggee JVM for information such as a stack trace or the value of a variable. Motivations for such a design have been discussed in detail in a previous paper [2].

To implement fair threads debugging, BUGLOO does not need special hooks in the Bigloo runtime. It only uses some breakpoint trickery and remote stack inspection. For instance, when the user single steps to the end of instant, the debugger sets a temporary breakpoint in scheduler code (somewhere in the Fair Threads runtime), and it tells the debuggee JVM to continue its execution. Later, when the execution breaks into the scheduler, the instant is over and the breakpoint is discarded. The same trickery is employed to implement the trace tool. Additionally, if *enhanced inspection* is required (see Section 3.1.2), the debugger inspects the debuggee's stack frame when such temporary breakpoints are reached, and stores additional information in the trace. An important property of this implementation is that the scheduling of fair threads is never affected by the debugger, since the scheduler is not aware of the instrumentation. In particular, the behavior of the debuggee program is not changed, as opposed to debugging tools that rely on a special interpreter or on source code instrumentation.

6.2 Benefits of the Tools

We have used our tools to debug BUGLOO itself. In the latest version of the debugger, we have modified the GUI so that it does not block anymore while the debugger queries information from the debuggee process. We have to manage a pool of fair threads, and to use techniques such as nesting schedulers. The trace tool helped us to understand that we were waiting a signal in the wrong instant. We also saw that fair threads cooperated too much, leading to superfluous empty instants in the execution.

The Fair Threads API and implementation are subject to change. The scheduler is likely to be re-implemented to avoid unnecessary context switches. For instance, all traces presented in this paper show that the control always returns to the scheduler before the end of instant. Our debugger will certainly help us in making a better implementation.

We believe that the tools we have presented can be very useful for educational purpose. They are simple to use and they help to un-

derstand what is going on in the scheduling of programs. This is a good means to get acclimatized with this style of concurrent programming based on signals and instants.

6.3 Current Limitations

We can detect dead-locks and live-locks, but the latter are currently difficult to deal with. For example, if a fair thread is stuck in an instantaneous loop, repeatedly awaiting a signal that is present in the instant, the trace records a lot of events and may grow too much to be displayed. To prevent this bug, the execution is automatically suspended when an abnormal number of events occurred during a single instant. Nevertheless, the repetitions stay visible in the trace and should be grouped.

Also, while the trace tool is a good way to visualize the communication of threads, it says nothing about the actual processing (like entering functions) done between the communication. This may make the trace difficult to read for programs that do a lot of computation and/or side effects in between synchronizations.

Finally, we have not provided yet a good support for visualizing large programs with many dozens of fair threads running concurrently. We should add means to show or hide signals and threads in the trace. Also, the ability to display long traces in multiple views in a Model-View-Controller fashion would be very useful.

7 Related Work

7.1 Debugging Concurrent Programs in Scheme

Every approach of concurrent programming comes with its specific problems with respect to debugging. Gambit-C 4.0 provides a user space implementation of preemptive threads *à la* POSIX based on continuations. We already discussed the problems inherent to this approach of concurrent programming. PLT's DrScheme [6] provides CML-like [15] concurrent primitives, where threads are meant to execute in independent address spaces with their only communication being via messages sent through channels. However this approach does not solve the problem of direct shared memory access. The thread system found in Scheme 48 [11] is based on optimistic concurrency, which provides a sort of per-thread cached view of the global address space. The use of caches makes it difficult to maintain a valid global state and to visualize it. FrTime [3] implements concurrency with functional reactive programming. It provides signal processors in the spirit of Fran [4] that run in response to "events" such as alarms or messages. To our knowledge, none of the former systems provides a complete tool for debugging concurrency.

In general, tools for debugging concurrent systems suffer from the same difficulty: one has to reason on a program by studying the order in which locks are acquired, or messages are passed. For example, the Concurrent Haskell Debugger [8] allows to visualize graphically the state of CML-like communication channels. The OptimizeIt! JVM profiler can log all the accesses to monitors that occur throughout the execution, to analyze them off-line. On the other hand, when debugging Fair Threads, one can reason on the full *algorithmic* logic of his program (*i.e*, context switches, end of instants, broadcasted/received signals), thanks to sequentiality, determinism and signals. Model checkers [10, 9] are one notable exception: these tools can exhibit complete sequences of execution that lead to a dead-lock or a live-lock. To achieve this, they use techniques such as temporal logics and state space exploration.

7.2 Advanced Traces Visualization

Traces are very effective to debug multi-threaded programs. In GThreads [22], Zhao and Stasko provide a complete set of trace views for graphically depicting the execution of program. One particularly interesting view is the so-called "History View", in which the lifetime of a thread is decomposed into colored segments which represent the functions entered by the thread. Our own trace tool would clearly benefit from this idea.

Jinsight [14, 19] is a Java tool for displaying and analyzing traces of programs. It can generate interactive views that can be unrolled or collapsed. It can also automatically detect patterns in the trace and group them to avoid cycles. We should integrate a similar mechanism into our scheduler traces, to fix the problem of live-lock tracing presented in Section 6.3.

8 Conclusion

In this paper we have presented an extension of the source-level debugger BUGLOO. It provides support for Fair Threads, a new thread-based concurrent programming framework that combines cooperative scheduling and strong communication based on synchronous reactive programming.

We showed that unlike the classic POSIX multi-threading approach, Fair Threads allow to provide the programmer with a strong debugging support. We have described three tools to deal with specific bugs that can arise with Fair Threads. First, an improved single-stepper. Second, a scheduler and signal inspector to analyze the state of threads when the program is suspended. At last, a scheduler tracer to analyze the progression of the scheduling off-line.

The presented tools are new in BUGLOO. We are working on faster ways of recording traces, and on other views that would give more insight on the scheduling activity. In the future, The Fair Threads framework will likely provide means to execute arbitrary computation asynchronously (*i.e*, in preemptive threads). We plan to extend the debugging support for these features.

9 References

[1] F. Boussinot. Java fair threads. Technical Report RR-4139, INRIA, 2001.

[2] D. Ciabrini and M. Serrano. Bugloo: A source level debugger for scheme programs compiled into jvm bytecode. In *Proceedings of the International Lisp Conference 2003*, 2003.

[3] G. Cooper and S. Krishnamurthi. Frtime: Distributed and asynchronous functional reactive programming. Technical Report CS-03-20, Department of Computer Science, Brown University, 2003.

[4] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, volume 32(8), pages 263–273, 1997.

[5] M. Feeley. Scheme request for implementation 18: Multithreading support. http://srfi.schemers.org/srfi-18/srfi-18.html, 2000.

[6] R. B. Findler, J. Clements, M. F. Cormac Flanagan, S. Krishnamurthi, P. Steckler, and M. Felleisen. Drscheme: A progamming environment for scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002.

[7] E. Gallesio and M. Serrano. Programming graphical user interfaces with scheme. *Journal of Functional Programming*, 13(5):839–866, September 2003.

[8] C. Grelck and S. Scholz. Axis Control in SaC. In T. Arts and R. Peña, editors, *Proceedings of the 14th International Workshop on Implementation of Functional Languages (IFL'02)*, volume 2670 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 2002.

[9] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder, 1998.

[10] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[11] R. A. Kelsey and J. A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1994.

[12] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, 1997.

[13] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming*. A Nutshell Handbook. O'Reilly & Associates, Inc., 1996.

[14] W. D. Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in Java. *Concurrency: Practice and Experience*, 12(14):1431–1454, 2000.

[15] J. Reppy. CML: A Higher-order Concurrent Language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*, number 6 in SIGPLAN Notices, pages 293–305. ACM Press, 1991.

[16] M. Serrano. Bee: an integrated development environment for the scheme programming language. *Journal of Functional Programming*, 10(4):353–395, 2000.

[17] M. Serrano, F. Boussinot, and B. Serpette. Scheme fair threads. In *To appear in the proceedings of the 6th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 2004.

[18] M. Serrano and P. Weis. Bigloo: A portable and optimizing compiler for strict functional languages. In *Static Analysis Symposium*, pages 366–381, 1995.

[19] G. Sevitsky, W. De Pauw, and R. Konuru. An information exploration tool for performance analysis of java programs. 2001.

[20] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39:1175–1185, 1990.

[21] R. Stallman and R. H. Pesch. *Debugging with GDB: the GNU source-level debugger*. Free Software Foundation, 4.09 for GDB version 4.9 edition, 1993. Previous edition published under title: The GDB manual. August 1993.

[22] Q. A. Zhao and J. T. Stasko. Visualizing the execution of threads-based parallel programs. Technical Report GIT-GVU-95-01, College of Computing, George Institute of Technology, 1995.

Acknowledgments

Many thanks to Bernard Serpette, Frédéric Boussinot, Manuel Serrano, Stéphane Epardaud, Florian Loitsch and to the anonymous reviewers for their helpful feedback on this paper. This document has been typeset in Skribe.