Debugging parallel programs using fork handlers

Javier Alcázar Zapién
International Systems Research Co.
Suginami 166-0003
Tokyo, Japan
javier@isr.co.jp

ABSTRACT

Nowadays multicore computers are easy to find everywhere, from mobile phones to high end servers. However, producing parallel programs that take advantage of these computers is not easy: parallel programs are error prone and finding these errors and their causes is a hard and time consuming task for which proper debugging is needed.

In the main implementation of Python, Ruby and other platforms, the only way to achieve real parallelism in multicore computers is using multiple processes rather than multiple threads. Debugging programs composed of multiple processes turns out to be more complicated than debugging multi-threaded programs composed of a single process.

This paper proposes using fork handlers to debug multi-process programs. To demonstrate this proposal we review the implementation for Ruby and Python in Dionea, an open source debugger. The same approach, and similar implementation techniques, can be extended to other platforms and debuggers lacking of debugging features for parallel programs that use processes to exploit concurrency on multicore computers.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Tracing; D.1.3 [**Concurrent Programming**]: Parallel programming

General Terms

Debugging, Parallel programming

Keywords

Debugging, multicore, parallel programming, Ruby, Python

1. INTRODUCTION

Programming languages have had concurrency constructs for decades [9] but, is just now with the ubiquity of multicore computers, which are easy to find everywhere, from mobile phones to high end servers, passing through network appliances, workstations and laptops, that parallel applications are in high demand.

Concurrent programs are more error prone than serial ones, and although there is decent debugging support for parallel programs written in C or Java, e.g., GDB/jdb + Eclipse, this is not the case for other mainstream languages such as Python and Ruby.

Python and Ruby provide thread constructs to express concurrency, but due to their Interpreter Locks, usually called GIL (Global Interpreter Lock) [3] in CPython and GVL (Global or Giant VM Lock) [18, 24] in CRuby, application threads cannot run in parallel, even on multicore computers. Therefore, on these platforms the only way to take full advantage of multiple cores is to use processes as concurrency construct but, the debugging support for multi-process parallel programs is limited.

Since multi-process parallel programs spawn processes using fork functions, we use a set of custom fork handlers, which allows to catch the debuggee processes, make the pertinent arrangements in both, child and parent processes, and continue the execution of the debuggees in a *low intrusive fashion* [26]. [1]

First, in order to present our proposal we make clear some concepts, and enumerate some of the reasons that make parallel programs more error prone. We also discuss the importance of having tools that aid the production of faster and correct parallel programs.

Next, we give an overview of Dionea and explain the propossed approach with its problems and solutions; followed by the implementation of the proposal in Dionea, which supports Python and Ruby. Then, we demonstrate simple and representative examples of parallel programs where Dionea has proved to be useful. Finally, we review Dionea's performance, compare some related contributions and discuss future work.

2. SERIAL, CONCURRENT AND PARALLEL

We use the terms *Processing Element* and *Unit of Execution* as in Patterns for Parallel Programming [17] where:

- **Processing Element (PE)** is a generic term for a hardware element that executes a stream of instructions, e.g., a core of a multi-core computer.

[1] low-intrusive refers to the capability of debugging a single thread while other threads continue executing freely.

PMAM '15, February 7-8, 2015, San Francisco Bay Area, USA

ACM 978-1-4503-3404-4/15/02 $15.00
http://dx.doi.org/10.1145/2712386.2712390

- **UE (Unit of Execution)** is a generic term for an executing entity, e.g., a process or a thread.

Through this text we differentiate three kinds of programs: serial, concurrent and parallel.

i **Serial** programs, are those that have been designed to execute their instructions one after another, i.e., only one instruction is executed at any given instant.

ii We call **concurrent** programs to those able to maintain active more than one task simultaneously. In concurrent programs that run on systems with a single PE, only one instruction is being executed at any given instant. Concurrent programs that share a common PE among all the given UEs only simulate the execution of more than one task at the same time. e.g., multithreaded programs running on a computer with a single core.

iii More than one instruction of a **parallel** program can be executed at the same time, i.e., while one instruction of a parallel program is executed in a PE, another instruction of the same program is being executed in another PE at the same instant. Parallelism can be seen as a special case of concurrency. With the appropriate hardware and software support concurrent programs can run in parallel.

3. FINDING ERRORS ON PROGRAMS IS NOT EASY, FINDING ERRORS ON CONCURRENT PROGRAMS IS DIFFICULT

A well known quote in the programming world says *Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it* [12]. This proves to be true in parallel programs, where actually it could be harder than twice. Concurrent programs are exposed to more errors than the traditional serial programs, errors such as, deadlock, liveliness, starvation and data race are unique to concurrent programs and are difficult to detect.

The Heisenberg's principle also relates to software debugging, the more examination is done to a program the more its normal behavior is altered. Therefore, is difficult to debug the exact behavior of a program without altering the behavior that it would show when is not being debugged, and concurrent programs are even more sensitive. Using log messages to debug might appear useful when lacking of proper debugging features but, in concurrent programs this practice may introduce more errors and hide the real problems; sometimes the streams or libraries to log messages have implicit locks or use other synchronization primitives that may interfere with the program execution; also executing the instructions to log messages alters the normal execution of the program.

On top of the previously expressed difficulties, most programmers have little or no experience in concurrent programming, which increases the error ratio on concurrent programs. Trying to find bugs is exhausting and time consuming, thus a tool such as Dionea is great help to produce correct parallel programs.

4. DIONEA OVERVIEW

Dionea has a distributed architecture following the client-server model [25, 26], this distributed architecture makes possible to debug multiple processes from a single client; the debuggee processes could be part of the same program or an independent processes.

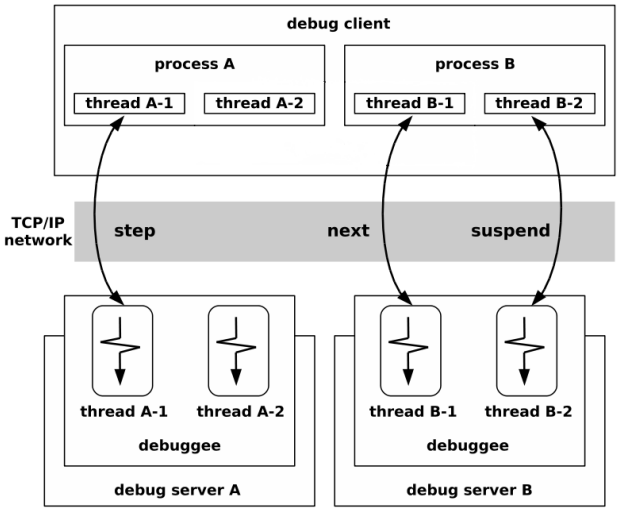

Figure 1: Dionea architecture

- **Debug server**: In Dionea, each debuggee has its own debug server, the debug server is a shim to control the execution of the debuggee based on the commands sent by the client. Both, debuggee and debug server run in the same process. The debug server traces debuggee's execution using custom functions in conjunction with the tracing facilities provided by the interpreters, i.e., Dionea's trace callback functions set by `Kernel#set_trace_func` and `sys.settrace` for Ruby and Python respectively. In Dionea, each debug server has a dedicated *listener thread* to receive requests and send responses from and to the client; this dedicated thread handles the requests asynchronously, treating each request as an event dispatched by a loop. The implementation of this listener thread is inspired by the Reactor pattern [27].

- **Client**: The client is an independent GUI that acts as interface between the user and the debuggee. Server and client interact through a predefined protocol using TCP/IP, making possible to debug remote processes.

- **Debug server - client communication**: Dionea uses three TCP/IP sockets for communication between the server and the client.

  1. One socket is used to listen and handle new connections.
  2. One more socket is used to synchronize the source code between the debug server and the client.
  3. Finally, another socket is used for sending debug commands. e.g., *set break point*, *continue*.

The client sends debug commands to the debugger server, such as *set break point*, *continue*, *step*, *next* and so on; the server receives commands from the client, executes them and sends appropriate responses to the client. Even when one single client is able to operate more than one process at the same time, the debuggee execution can be controlled individually for each process or even each thread, this is what makes Dionea a low-intrusive debugger [26], however, Dionea can also operate over the whole program, e.g., suspending all the threads of a multithreaded program.

4.1 Debug sessions

In Dionea, a debug session is a sequence of interactions between debugger and debuggee, i.e., user commands sent from the GUI client to the debug server, and replies sent from the debug server to the client. Figure 1 shows a single client controlling the execution of more than one debuggee, which implies that one client maintains one session per debuggee. It is analogous to a single web browser having different web sessions on different web pages, e.g., a web browser maintains a session with a web based mail client to check user's email, and at the same time another session is maintained to access online bank. However, in Dionea's case a debug server is tied to a single client, otherwise two different clients could control the same debuggee at the same time, making it inconsistent, which obviously is an undesired situation. The relationship between Dionea client and servers is in the form $1client : Nservers; 1server : 1client$.

4.2 Debug views

Debug views exist in the context of debug sessions, debug views can be understood as the sequence of interactions between the client and a concrete UE of the debuggee as shown in Figure 3. There is only one debuggee view active at a time. Debug views are presented on the client side in form of source code and variables with their values, and the command shell or the buttons on the toolbar serve as terminal to send commands to the UE being debugged.

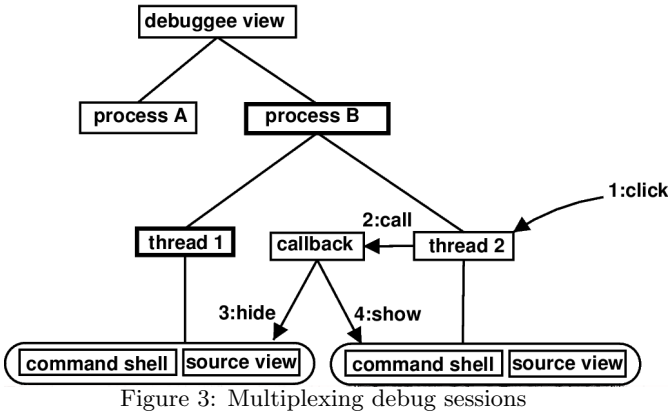

Figure 3: Multiplexing debug sessions

In Figure 3 there are two sessions, one between the client and *process A*, and other between the same client and *process B*. In this scenario, the *thread 1* of *process B* is active, now let's examine what happens to the *thread 2* of *process B* when:

1 *thread 2* of *process B* is clicked in the GUI.

2 The previous action generates a call, which in turn triggers the execution of the trace *callback*.

3 Source of *thread 1*, which until now was active is hidden on the client.

4 The debuggee view corresponding to *thread 2* becomes active, therefore source of *thread 2* is displayed on the client.

Summing up, debug sessions are between the client and debuggee processes, and debuggee views are between the client and debuggee threads.

5. FORK

5.1 Fork functions

Is common that libraries to write parallel programs, use fork functions to create processes that will act as UEs to exploit parallelism. Fork functions usually make a copy from the actual process. General examples of fork functions include: the function `fork (2)` defined in `unistd.h` [28], `clone (2)` defined on `sched.h` which is invoked on Linux systems instead of `fork` [13], and `os.fork` [30] on Python. A fork function can be either a system call or functions provided by the relevant language, in the later case these functions may in turn use system calls.

When using fork functions is common to call exec functions in the child process right after fork, doing so replaces the process image, but this is not the case when using processes as means of concurrency. Hence forking without calling exec is an special case that requires special treatment; also is not recommended to fork multithreaded process, therefore when it becomes necessary (as in the case of Dionea, which is explained later), special precautions should be taken. In Python and Ruby, fork semantics indicate that, only the thread that called fork remains in the child, i.e., other threads present in the parent at the moment of fork remain in the parent but not in the child.

5.2 Fork handlers

Fork handlers are functions hooked to the fork function, therefore, when the fork function is called these hooked functions are called. Since using fork functions to create new processes is a standard way to create processes, and using fork handlers to decorate fork functions can be applied generally, we believe that using fork handlers to debug process based parallel programs is a sound approach.

In Dionea's case, fork handlers are responsible to ensure the proper execution and debugging around the fork event; they should take care of the parent process before the fork occurs, during and after the fork in both, the parent and the child process. When designing and implementing fork handlers, it should be noted that other hooked fork handlers will be called along with our fork handlers.

In POSIX programming, fork handlers are registered (hooked) with the `pthread_atfork()` function [28], other interesting examples of fork handlers can be found in MRI (Matz's Ruby Interpreter, is de facto implementation of Ruby 1.8) and YARV [23] (Yet Another Ruby VM is de facto implementation of Ruby 1.9.).

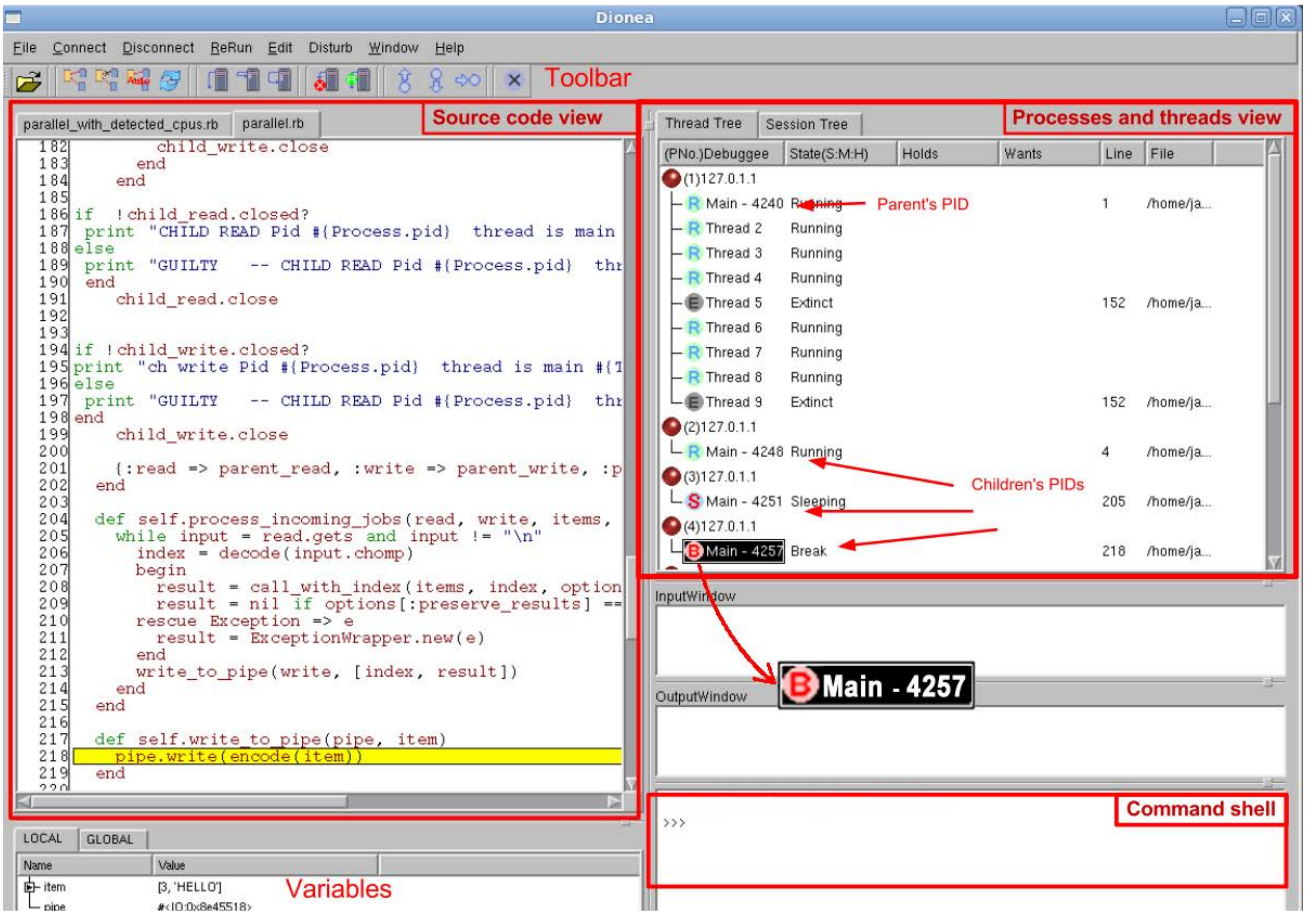

Figure 2: Dionea client

Source code view: In the figure above, the process *(4)* with PID *4257* is selected in the *Process and threads view*, therefore the source code corresponding to process *(4)* is shown here. In this situation, the active *debug view* corresponds to the main and only thread of process *(4)*, which becomes active.

Processes and threads view: This area shows processes and their threads. Whenever a thread or process is clicked its corresponding *debug view* becomes active, as a consequence its source code is shown in *Source code view*.

Command shell: The command shell is used to send commands to the debuggee, e.g., *continue*, *step*, *next*.

Input window: This area corresponds to the standard input of the active debug view, if the program requires input from the user, this is the place to enter data.

Output window: This area corresponds to the standard output of the active UE.

Variables: Variables and their values are displayed in the area just below of the *Source code view*.

```
13176 void
13177 rb_thread_atfork()
13178 {
13179     rb_thread_t th;
13180
13181     rb_reset_random_seed();
13182     if (rb_thread_alone()) return;
13183     FOREACH_THREAD(th) {
13184         if (th != curr_thread) {
13185             rb_thread_die(th);
13186         }
13187     }
13188     END_FOREACH(th);
13189     main_thread = curr_thread;
13190     curr_thread->next = curr_thread;
13191     curr_thread->prev = curr_thread;
13192     STOP_TIMER();
13193 }
```

Listing 1: Fork handlers in MRI ruby-1.8.7-p358, `eval.c` [16].

Fork handlers shown in listings 1 and 2 are used to enforce the *only the thread that called fork exists in the child* semantics, and take care of synchronization objects. Although Dionea uses fork handlers to debug parallel programs composed of multiple process, the fork handlers shown in listings 1 and 2 are relevant to Dionea because they will be executed along with Dionea fork handlers. The fork handlers presented here take care of synchronization objects, therefore Dionea's may choose to don't take care of this. However, since this is not stated in the Ruby specification [10] and other Ruby implementations may not take care of synchronization objects, Dionea fork handlers also take care of synchronization objects before forking.

```
2745 static void
2746 rb_thread_atfork_internal(int (*atfork)
      (st_data_t, st_data_t, st_data_t))
2747 {
2748     rb_thread_t *th = GET_THREAD();
2749     rb_vm_t *vm = th->vm;
2750     VALUE thval = th->self;
2751     vm->main_thread = th;
2752
2753     native_mutex_reinitialize_atfork
         (&th->vm->global_vm_lock);
2754     st_foreach(vm->living_threads,
         atfork, (st_data_t)th);
2755     st_clear(vm->living_threads);
2756     st_insert(vm->living_threads,
         thval, (st_data_t)th->thread_id);
2757     vm->sleeper = 0;
2758     clear_coverage();
2759 }
```

Listing 2: Fork handlers in YARV ruby-1.9.2-p180, `thread.c` [18].

5.3 Dealing with forks in Dionea

The debug server always will have at least two threads, i.e., the debuggee's main thread and the debug server listener thread. The semantics of fork functions in POSIX [28], Ruby [32] and Python [30] are that only the thread that called fork survives in the child process. These semantics don't apply everywhere; for example, in Scsh all application threads are copied into the child when a process is forked [6].

Dionea's fork handlers should be aware of this situation. Whenever a debuggee (with its debug server) creates another process, these handlers ensure the proper debugging and tracing of the spawned process; they deal with three main problems.

1 **Ensuring the new process continues running**. We know that in Ruby and Python only the thread calling fork survives in the child. It is also known that concurrent programs use synchronization objects like mutexes, condition variables, etc. So, Dionea takes ownership of the debuggee's synchronization objects, e.g., `mutex.lock` before forking the process. Taking ownership of the synchronization objects ensures that the thread that survives in the child owns the synchronization objects, therefore this thread can later release the synchronization objects, eliminating the possibility of deadlocks.

2 **Debugging on child**. As depicted in figure 4 when a child process is created, it inherits the data structures from its parent, these data structures contain metadata for debugging, such as breakpoint information, PID (process identifier), debug session and so on. These data structures don't contain child information but parent information, therefore they should be updated with child's information.

3 **Establishing proper communication with the client**. When a child process is created, it inherits the sockets from its parent and will try to communicate with the client using its parent sockets, this would result in mixed requests and responses. Therefore, the child needs to establish its own debug session with the client using its own sockets. Dionea's fork handlers use a temporary file, where the port number of the most recently created process is saved. See figures 5 and 6.

In figure 4, debug sessions and other metadata are in the *data structures* block, these data structures are inherited in the child process from its parent, the child process needs to get rid of its parent data structures and create its own. Also note that, in the child process all the threads disappear except for the *thread that called fork*; in order to continue debugging on the child, Dionea debug server needs to establish a session with the client, to accomplish this, the *listener thread* is recreated in the child.

5.4 Fork handlers in Dionea

To cope with the situations described in the previous section, Dionea uses a set of augmented fork functions which in turn call Dionea fork handlers.

These augmented fork functions do the following:

A **Prepare fork**. Acquire control over synchonization objects. Disable the tracing until the listener thread is restarted, to avoid a deadlock in the child process, therefore is not possible to step inside of the augmented fork.

B **Handle parent at fork**. Immediately after the fork, release control of synchronization objects, and re-enable tracing.

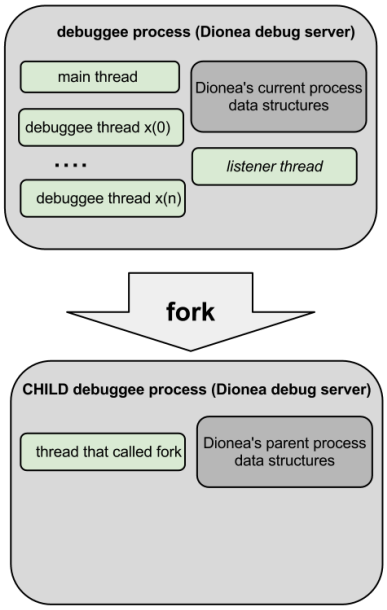

Figure 4: Dionea debuggee before and after fork

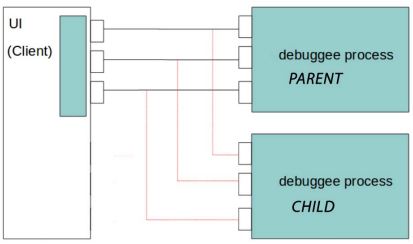

Figure 5: When a new child is created, at the beginning its sockets are the same as its parents'

C **Handle child at fork**. Initialize the synchronization objects, close the inherited sockets, initialize the data structures, create a listener thread, register the thread that called fork as the main thread, inform the client about the creation of a new debuggee, and finally re-enable the tracing that was disabled in **A**.

In Ruby's case Dionea uses metaprogramming techniques to catch fork events. The original Ruby methods of `Process#fork` and `Kernek#fork` have been modified to handle debugee's fork using the Ruby's feature of open class [21].

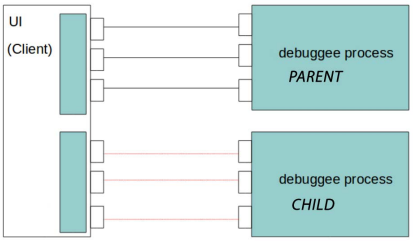

Figure 6: Dionea's fork handlers create appropriate sockets for the child debuggee

```ruby
module Kernel
  alias_method :old_fork, :fork

  def fork(&block)
  Dionea.prepare_fork #A
    pid = old_fork

    #Executes child's code
    if pid == nil then
    Dionea.handle_child_atfork #C
      if block_given? then
        # executes block's code
        yield
        #free resources, inform termination
        Dionea.debugger.at_finalize_proc
        # terminates the process as specified
        #  by the documentation
        Kernel.exit(0)
      else
        return nil
      end
    end#if
    Dionea.unlock_parent  #B
    Dionea.processes << pid  unless
      Dionea.processes.include?(pid)
    return pid
  end
end#Kernel
```

Listing 3: Dionea's Ruby fork.

In Python's case Dionea uses a method alias, so when the original method `os.fork` is called the Dionea's method is executed instead.

```python
def _dionea_fork():
    #prepare_fork: lock the mutex in dionea
    handle_prepare_fork() #A
    pid = __python_fork()
    if pid == 0:
        #Execute child code
        handle_child_at_fork() #C
        return 0
    else:
        handle_parent_at_fork() #B
        # purpose: INT --> shutdown
        _processes.append(pid)
        return pid

__python_fork = os.fork

os.fork = _dionea_fork
```

Listing 4: Dionea's Python fork.

6. USAGE SCENARIOS

6.1 Typical usage of Dionea

Once Dionea concepts have been outlined, let's describe a typical Dionea's use case. First, we start Dionea server issuing `ruby bin/dioneas.rb path/to/debuggee/ruby/program.rb` in Ruby's case. And `python dioneas.py path/to/debuggee/`

`python/program.py` for Python programs; once Dionea server has been started it waits until the client connects to it. Then the client sends debug commands to the debug server, e.g., *set breakpoint, continue*, either via the command shell or clicking on the source and toolbar's buttons; the server receives commands from the client, executes them and sends appropriate response to the client. Using low-intrusive features, i.e., being able to debug individual processes while simultaneously other processes continue running, is more efficient than stopping all the processes because the overhead associated to debugging only affects particular processes.

6.2 Finding deadlocks in Ruby programs

In Ruby as in other languages, concurrent programs use synchronization mechanisms to control UEs and data, however they should be used very carefully because a simple mistake can lead to subtle errors, such as deadlocks.

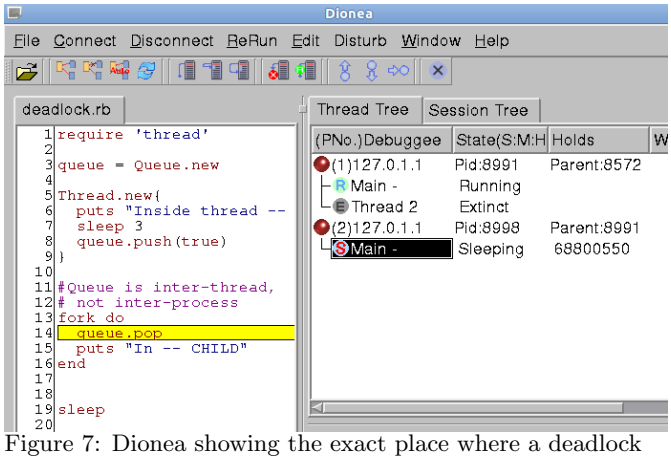

Figure 7: Dionea showing the exact place where a deadlock occurs.

```ruby
require 'thread'

queue = Queue.new

Thread.new{
  puts "Inside thread -- PARENT"
  sleep 3
  queue.push(true)
}

#Queue is inter-thread,
# not inter-process
fork do
  queue.pop
  puts "In -- CHILD"
end

sleep
```

Listing 5: Intentional deadlock

In figure 7 we have intentionally produced a deadlock, Dionea shows the line number where the deadlock has occurred. Without Dionea only a message like the shown in listing 6 would be displayed, such message is detailed but not clear to find where the deadlock occurred, especially in real applications where the exact place where the deadlock occurred may not be present in the stack trace.

```
Inside thread --
thread.rb:185:in 'sleep': deadlock detected (fatal)
   from thread.rb:185:in 'block in pop'
   from <internal:prelude>:10:in 'synchronize'
   from thread.rb:180:in 'pop'
   from deadlock.rb:14:in 'block in <main>'
   from deadlock.rb:13:in 'fork'
   from deadlock.rb:13:in '<main>'
```

Listing 6: Standard deadlock error message in Ruby

6.3 MapReduce in Python

In CPython threads can not run in parallel, however since version 2.6 the multiprocessing package subtitled *Process-based "threading" interface* [31] is part of the standard distribution, and is available as egg (Python library) for previous versions. This library provides an API very similar to threads, allowing to write programs that can take advantage of multicore computers using processes rather than threads to express concurrency. The ability of Dionea to debug over multiple processes fits very well for programs using Python's multiprocessing library.

Figure 8 shows a snapshot of tracing a word count program by MapReduce, the parent and the worker processes share the same input and output queues. The queue is implemented using a semaphore and a pipe. Functions or methods to be executed by the child process are passed from parent to child via queues encoded using pickle. When every other process is stopped by break points as shown in Figure 8, we observe that an available child process takes over the jobs. The low intrusive operations for threads and processes make it easy and efficient to analyze the behavior of parallel programs.

6.4 Finding errors in Ruby libraries

Ruby libraries are usually packed in a defined format called gems, there are different gems to create parallel programs based on process, a popular one is *parallel* [8]. The parallel gem spawns workers, either threads or processes, assigning tasks to them and getting their results. When processes are used the communication is done via `IO.pipe`.

When Dionea debugs parallel programs using the version 0.5.9 of the parallel gem, where `fork` and `IO.pipe` operations take place interleaved by the threads that interact with the child processes, Dionea very often detects a concurrency error that rarely happens running without Dionea: The debuggee processes get into a deadlock situation due to the failure in closing input pipe of the child process. The discussions with the developers resulted in the fruitful upgrade to 0.5.10 and to 0.5.11. All the unnecessary pipes used for each of the forked processes are copied. Therefore, the forks must be done sequentially by the main thread, not by the threads that interact with the child processes. By doing so, each of the forked processes can close the copied but unused pipes (for sibling processes). Once the problem has arisen, is easy to reproduce such critical cases of concurrency, and moreover identify the cause: setting *disturb mode* in Dionea, which will cause to stop the execution of every newly created process or thread; and then interleaving the execution

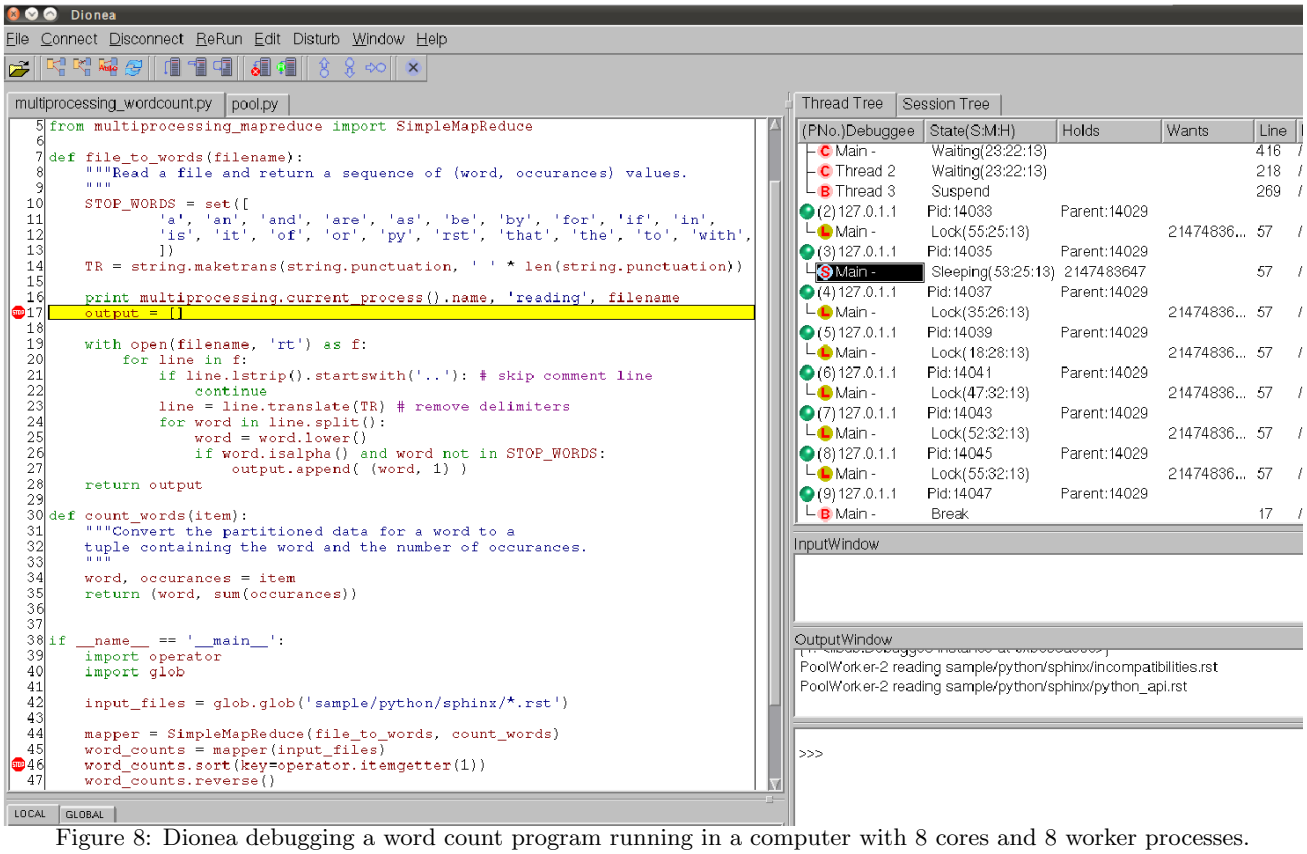

Figure 8: Dionea debugging a word count program running in a computer with 8 cores and 8 worker processes.

of the threads using Dionea's low intrusiveness.

7. PERFORMACE

A Python program that uses multiprocessing [31] to implement MapReduce [15] was prepared to quantify the overhead of running a program with Dionea and no breakpoints. This program *maps* words that contain only letters and are not reserved words, then the program *reduces* the values obtained in the map phase to calculate the frequency of each word.

An increment of 12.11% in the execution time was found for a small set of data when executing the program with Dionea, while bigger sets of data showed an increment of around 20%.

CPU	Intel(R) Core(TM) i5 CPU, 4 cores
HD	OCZ Technology Vertex 2 SATA II (SSD)
Memory	6GB DDR3 1333MHz
OS	Ubuntu 13.04 (3.8.0-27 SMP x86_64 GNU/Linux)
Python	2.5.2

Table 1: Computer specifications

The same program was also run in the same way for Rust's source code (master 7613b15). The average time without Dionea was 3'49" and with Dionea was 4'36".

Running a program with a debugger attached and no breakpoints, as we have done in this section, is not a common scenario but, it has been useful to quantify the overhead when debugging with Dionea.

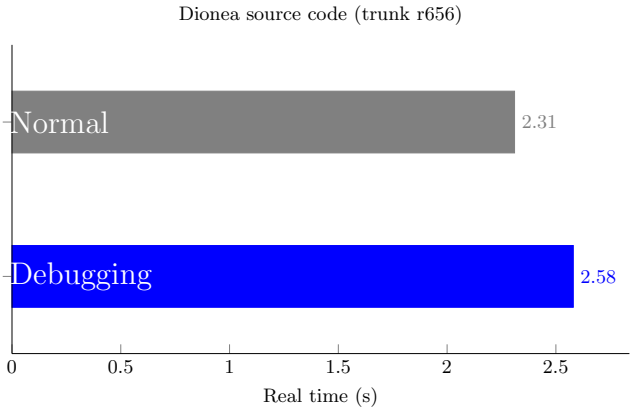

Figure 9: Calculating words' frequency with Dionea in Dionea source code showed an increment of around 12%.

8. RELATED WORK

8.1 Debuggers for concurrent software

The well known GNU Debugger (GDB), is widely used to debug native programs, and since version 7 has been able to debug programs that create additional processes using the fork functions [5], this feature would allow us to debug the CRuby or CPython interpreters, but would be very difficult to debug the programs that are being run in the interpreters, i.e., we can debug the CRuby interpreter itself but not directly the Ruby program. GDB uses the trace functionality provided by the operating system, i.e., `ptrace` and signals,

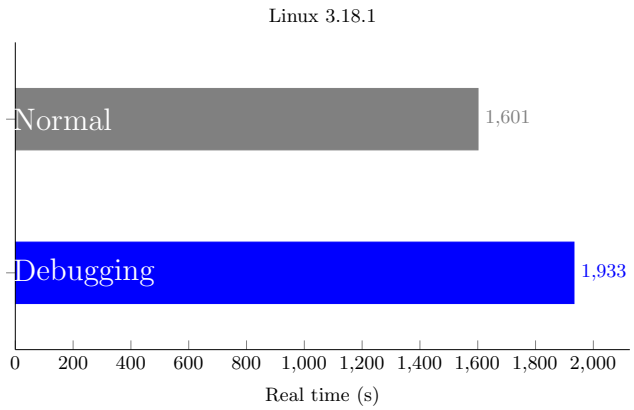

Figure 10: Calculating words' frequency with Dionea in Linux source code showed an increment of around 20%.

whereas Dionea uses the trace functionality provided by the interpreters.

DBX is another popular debugger for UNIX systems, which in conjunction with Oracle Solaris Studio is able to debug programs that create other processes using fork functions. We can infer that the techniques used by DBX to debug forked processes are similar to those used in GDB.

It has been possible to debug concurrent Java programs for a while, either in high or low-intrusive fashion, Java debuggers rely on the tracing support provided by the JVM (Java Virtual Machine). Should be noted that the concurrency models used by CPyton, CRuby and popular JVM's differ significantly. For example JVMs usually allow threads to run in parallel, therefore there is no need to create additional processes to achieve parallelism.

There are other debuggers capable to debug concurrent and parallel programs, some of them have very interesting features, and different approaches have been used to implement those features. An incomplete list of these debuggers includes: STAT (Stack Trace Analysis Tool), a tool that uses stack trace analysis techniques, intended for MPI programs [1]; KDB, a multi-threaded debugger for multi-threaded applications [4]; TotalView, a debugger that allows debugging multiple processes from a single GUI, it targets Fortran, C and C++ programs [14].

8.2 Debuggers for Python and Ruby

PyCharm is a Python specific IDE that added support for debugging multi-process programs around the same time Dionea did [11, 2]. Even though the features for debugging multi-process in Dionea and PyCharm have been developed independently, the underlying principles are similar, i.e., client-server model and fork handlers. While Dionea and PyCharm have similar features, PyCharm is a commercial product developed by JetBrains, which also has a free and open source version; whereas Dionea has been developed as a research exercise in the Kanazawa Institute of Technology. A similar implementation to debug parallel multi-process programs in PyCharm, which is a commercial product, proves that our approach is solid and generally applicable.

`pdb`, the debugger provided with CPython comes in handy for debugging serial programs. In the Ruby side there have been different debuggers, some of them work only for MRI and some others only for YARV, whereas Dionea works with both.

Dionea is also able to debug parallel programs using JRuby[2], JRuby[19] is a Ruby implementation in Java, which means that JRuby programs run on top of the JVM. Multithreaded JRuby programs can run in parallel exploiting the advanced threading support in the JVM and Dionea is able to debug these programs.

Dionea also supports Ruby Enterprise Edition, which is a modified version of MRI that has modified its fork function implementing copy-on-write semantics [22].

9. CURRENT STATUS AND FUTURE WORK

Dionea GUI uses Qt3 and CPython 2.5, which are considered obsolete; therefore it is desirable to update Dionea GUI to Qt4 or refactor it as an Eclipse plugin and add support for newer versions of CPython.

There is previous research on debugging programs that use Hardware Transactional Memory (HTM) [33, 7] and it has been proved that is possible to eliminate the GVL of CRuby using HTM [20]. These facts suggest that it would be possible to add support in Dionea for debugging parallel Ruby programs that use HTM instead of GIL.

Other areas of future work include: support for Jython, and support for PHP, which does not support threads, leaving processes via the PCNTL functions as the only way to achieve concurrency in PHP [29].

10. CONCLUSIONS

We have described how fork handlers can be used to debug parallel programs on multicore computers, and demonstrated the implementation in Dionea for two different platforms. This approach proves to be a general solution, that so far has been applied for CPython, MRI, YARV and could be extended to other platforms.

Understanding the internals of the libraries to create parallel programs in Ruby and Python, the behavior of the CPython and CRuby interpreters in multicore computers and the semantics of fork handlers, new features have been added in Dionea to debug parallel programs that run on multicore systems. We have demonstrated these features with use cases for Python and Ruby programs.

11. ACKNOWLEDGMENTS

Thanks to Prof. Norio Sato. This work would not have been possible without his insightful guidance.

The initial part of this work was supported by the Programa para la Formación de Recursos Humanos en la Asociación Estratégica Global México Japón.

Thanks to the anonymous reviewers.

12. REFERENCES

[1] D. H. Ahn, B. R. de Supinski, I. Laguna, G. L. Lee, B. Liblit, B. P. Miller, and M. Schulz. Scalable temporal order analysis for large scale debugging. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 44:1–44:11, New York, NY, USA, 2009. ACM.

[2]While adding support for JRuby a major bug was found in the core classes of JRuby, we provided a patch that was incorporated in JRuby 1.6.5.

[2] J. Alcázar. Dionea's SVN revision 525. *Everything seems to be ok for python multiprocessing package*. October 2011.

[3] D. Beazley. Inside the Python GIL. Python Concurrency Workshop (Chicago). 2009.

[4] P. A. Buhr, M. Karsten, and J. Shih. Kdb: A multi-threaded debugger for multi-threaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, SPDT '96, pages 80–87, New York, NY, USA, 1996. ACM.

[5] Free Software Foundation, Inc. Debugging Forks. In *Debugging with GDB*, chapter 4.1. 2012.

[6] M. Gasbichler and M. Sperber. Integrating User-Level Threads with Processes in Scsh. *Higher Order Symbol. Comput.*, 18(3-4):327–354, Dec. 2005.

[7] J. E. Gottschlich, R. Knauerhase, and G. Pokam. But how do we really debug transactional memory programs? In *Presented as part of the 5th USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, 2013. USENIX.

[8] M. Grosser et al. parallel gem. 2011.

[9] P. B. Hansen, E. W. Dijkstra, and C. A. R. Hoare. *The Origins of Concurrent Programming: From Semaphores to Remote Procedure Calls*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[10] ISO/IEC. *Ruby Standardization, ISO/IEC 30170:2012*. ISO/IEC, 2001.

[11] D. Jemerov. New PyCharm 2.0 EAP build: Django i18n, multiprocess debugging. October 2011.

[12] B. W. Kernighan and P. J. Plauger. *Elements of Programming Style*. McGraw-Hill, Inc., New York, NY, USA, 1974.

[13] M. Kerrisk et al. CLONE(2). Linux man-pages project release 3.35. 2011.

[14] B. Kingsbury. Organizing processes and threads for debugging. In *Proceedings of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging*, PADTAD '07, pages 21–26, New York, NY, USA, 2007. ACM.

[15] R. Lämmel. Google's mapreduce programming model - revisited. *Sci. Comput. Program.*, 68(3):208–237, Oct. 2007.

[16] Y. Matsumoto. eval.c. MRI Implementation 1.8.7. 2012.

[17] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004.

[18] N. Nakada, K. Sasada, et al. thread.c. YARV Implementation 1.9.2-p180. 2011.

[19] C. O. Nutter, T. Enebo, N. Sieger, O. Bini, and I. Dees. *Using JRuby: Bringing Ruby to Java*. Pragmatic Bookshelf, 1st edition, 2011.

[20] R. Odaira, J. G. Castanos, and H. Tomari. Eliminating global interpreter locks in ruby through hardware transactional memory. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 131–142, New York, NY, USA, 2014. ACM.

[21] P. Perrotta. *Metaprogramming Ruby*. Pragmatic Bookshelf, 1st edition, 2010.

[22] Phusion^{TM}The computer science company. What is ruby enterprise edition?. *Ruby Enterprise Edition FAQ*. October 2013.

[23] K. Sasada. YARV: Yet Another RubyVM: Innovating the Ruby Interpreter. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 158–159, New York, NY, USA, 2005. ACM.

[24] K. Sasada. Implementation of Ruby 1.9.3 and later. RubyConf 2011. 2011.

[25] N. Sato and K. Kosuga. Session-aware debugging features for web applications using ruby and python frameworks. In *Workshop on Domain Specific Approaches to Software Test Automation: In Conjunction with the 6th ESEC/FSE Joint Meeting*, DOSTA '07, pages 13–19, New York, NY, USA, 2007. ACM.

[26] N. Sato, K. Nagai, Y. Itoh, M. Ogura, and K. Kosuga. Low-intrusion Debugger for Python and Ruby Distributed Multi-thread Programs. *IPSJ Journal*, 2004.

[27] D. C. Schmidt. Using design patterns to develop reusable object-oriented communication software. *Commun. ACM*, 38(10):65–74, Oct. 1995.

[28] The IEEE and The Open Group. *The Open Group Base Specifications Issue 6 IEEE Std 1003.1*. 2004.

[29] The PHP Group. PCNTL Functions. In *PHP Manual*. 2012.

[30] The Python Software Foundation. Miscellaneous operating system interfaces. In *The Python Documentation*, chapter 15.1. 2012.

[31] The Python Software Foundation. multiprocessing - Process-based "threading" interface. In *The Python Documentation*, chapter 16.6. 2012.

[32] The Ruby Community. Module: Kernel. In *Ruby-Doc.org*. 2012.

[33] F. Zyulkyarov, T. Harris, O. S. Unsal, A. Cristal, and M. Valero. Debugging programs that use atomic blocks and transactional memory. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 57–66, New York, NY, USA, 2010. ACM.