Detecting Data Races in Sequential Programs with *DIOTA*

Michiel Ronsse, Jonas Maebe, and Koen De Bosschere

Department ELIS, Ghent University, Belgium
{michiel.ronsse,jonas.maebe,koen.debosschere}@UGent.be
http://www.elis.UGent.be/diota/

Abstract. In this paper we show that data races, a type of bug that generally only causes havoc in parallel programs, can also occur in sequential programs that use signal handlers. Fortunately, it turns out that adapting existing data race detectors to detect such bugs for sequential programs is straightforward. We present such a tool, and we describe the modifications that were necessary to detect data races in sequential programs. The experimental evaluation revealed a number of data races in some widely used programs.

1 Introduction

Developing parallel programs is encumbered by the fact that proper synchronisation must be used to access shared variables. A lack of synchronisation will lead to a data race [1]: two or more parallel executing threads access the same shared memory location in an unsynchronised way, and at least one of the threads modifies the contents of the location. Note that the prerequisite *parallel execution* does not imply that the application should be executed on a parallel computer with multiple processors on which true physical parallel execution is possible. A data race is also possible on computers with only one processor, as the scheduler simulates logical parallellism between the threads.

As data races are (most of the time) considered bugs, they should be removed. Fortunately, automatic data race detection is possible and a number of data race detectors have been build in the past [2, 3]. Basically, two types of information are needed to perform data race detection: a list of all load/store operations (with information about the memory address used, the thread that executed the instruction, the type of operation (load, store or modify)) and information about their concurrency. The latter information can be deduced by intercepting the synchronisation operations (e.g. thread creation, mutex operations, semaphore operations, ...).

Developers of sequential programs do not have to worry about data races, except if the program uses signal handlers. We'll show that signal handlers introduce *logical* parallellism and as such data races can show up.

In the remainder of this paper, we start with a short description of UNIX type signals and signal handlers. Next, we'll show that signal handlers indeed

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 82–89, 2004.

introduce parallellism, even in sequential programs, and we'll describe how an existing data race detector for IA32 binaries running on Linux has been adapted to detect them. We end with an experimental evaluation. To the best of our knowledge, there are no other data race detectors that deal with signal handlers.

2 Signals

A signal is an important communication mechanism that was already supported by the very first UNIX-implementations [4]. A signal is basically a message (signal number, identity of the sender, context information, ...) that can be sent to a process. The signal can either be sent by a process (including the recipient) using the `kill` system call, or the kernel can trigger one upon the occurrence of a certain event (e.g. a processor trap). In both cases, the kernel will notify the application by sending the appropriate signal (read: setting a bit in the process context). The next time the application is scheduled, the application will execute the corresponding signal handler. Such a signal handler is a function that is registered with the kernel by means of the `signal` or `sigaction` system call. E.g. Figure 1 shows a program that will print `You pressed ^C!` each time `^`C is pressed.

Typical signals are SIGSEGV (the application tries to access an invalid memory location), SIGILL (an illegal instruction was executed), SIGFPE (a floating point exception occurred), SIGPIPE (broken pipe), SIGALRM (normally sent by an application to itself in order to execute a function at a certain time in the future), SIGINT (interrupt signal sent by the shell when you press `^`C), SIGUSR1 (a general purpose signal), SIGWINCH (sent by the window manager when the size of the window changes), ...

As mentioned above, when a signal reaches[1] an application, the normal program execution is interrupted and a *signal handler* is executed instead. As the program execution can be interrupted at any time, esp. if a signal was sent by another application, a data race can occur if one does not pay attention to this fact. Figure 2 shows an example: in `main()` the variable `global` is incremented and decremented a (large) number of times, resulting in a final value of 0 for `global`. However, each time `^C` is pressed, a signal handler gives `global` an additional increment, resulting in a positive final value for `global`. This should be considered to be a data race: two unsynchronised write operations to the same variable. Of course, this is an artificial example, but similar problems can occur in real life applications, e.g. a graphical application that receives a SIGWINCH signal should update its width and height with care.

In [5], a number of Linux applications were examined, and the author states that "80 to 90% of signal handlers we have examined were written in an insecure manner". The manual inspection revealed races in such high-profile applications as `sendmail`, `WU-FTP` and `screen`.

[1] Two signals (SIGKILL and SIGSTOP) never reach an application, but are directly handled by the kernel.

```
#include <signal.h>

void sigint(){puts("You pressed ^C!");}

main(){
  signal(SIGINT, sigint);
  while(1){}
}
```

Fig. 1. A simple signal handler example.

```
#include <signal.h>

#define N 1<<24
unsigned global=0;

void sigint(){global++;}

main(){
  unsigned i;
  signal(SIGINT, sigint);
  i=N; while (i--) global++;
  i=N; while (i--) global--;
}
```

Fig. 2. A small program exhibiting a data race involving `global`.

3 Dealing with Signals in the Context of Data Race Detection

In order to detect data races in signal handlers, one has to trace the memory operations and gather information about their concurrency. The latter requires some attention: although signal handlers normally do not use synchronisation operations in order to prevent data races, not all signal handler memory operations are parallel with all memory operations executed by the application itself.

Attention should be paid to the following points:

- the data race detector should take into account that it is impossible to execute a signal handler before the signal handler is registered with the kernel. E.g. Figure 3 shows a program in which `global` is modified in `main()` and in the `sigint()` signal handler. As the signal is installed after the modification of `global` in `main()`, the accesses to `global` will never be executed in parallel, and hence no data race can occur.
- the above also applies to a signal that was sent by the application itself, e.g. using the `kill()` or `alarm()` function. Figure 4 shows an application that sends an alarm signal to itself. In this case, no data race occurs as the

```
#include <signal.h>

unsigned global=0;

void sigint(){global++;}

main(){
  global++;
  signal(SIGINT, sigint);
  sleep(10);
}
```

Fig. 3. This program does not exhibit a data race involving `global` as `main` accesses `global` before the signal handler is installed.

```
#include <signal.h>

unsigned global=0;

void sigalarm(){ ❹
  global+;+
} ❸❺

main(){
  signal(SIGALRM, sigalarm); ❶
  global+;+
  alarm(10); //send a SIGALRM to ourself in 10 seconds ❷
}
```

Fig. 4. This program does not exhibit a data race involving `global` as `main` accesses `global` before sending the signal. However, a data race will occur if another process also sends a `SIGALRM` to the application. The numbers refer to the modifications that were applied to our data race detector, see section 4.3.

`alarm()` call always precedes the execution of the `sigalarm()` handler. Of course, as the `SIGALRM` can also be sent by another application (resulting in a data race), the signal context should be checked; if the sender is the process itself, the data race detector should update the concurrency information.

- the easiest way to prevent data races in signal handlers is blocking the arrival of signals. Signals can be blocked automatically each time a certain signal handler is executed or explicitly using the `sigprocmask` system call. For both methods, a mask with one bit per signal is registered in the kernel. A data race detector should take this blocking into account.
- during the execution of a signal handler, the thread executing the signal handler should be assigned a new thread number. This also applies to a signal handler preempting the execution of another (or the same) signal handler.

4 Implementation in *DIOTA*

We have implemented data race detection for sequential programs in the data race detector that is part of our *DIOTA* (Dynamic Instrumentation, Optimisation and Transformation of Applications) framework for Linux running on IA32 processors [6].

4.1 Description of *DIOTA*

DIOTA is implemented as a shared library for the Linux/80x86 platform. It instruments programs at the machine code level, so it is completely compiler- and language-agnostic and can also cope with hand-written assembler code. It has support for extensions to the 80x86 ISA such as MMX, 3DNow! and SSE and is written in an extensible and modular way so that adding support for new instructions and new types of instrumentation is easy.

An environment variable is used to tell the dynamic linker to load the *DIOTA* library whenever a dynamically linked application is started[2]. An `init` routine allows *DIOTA* to be activated before the main program is started, after which it can gain full control and start instrumenting.

The instrumentation happens gradually as more code of the program (and the libraries that it uses) is executed, so there is no need for complex analysis of the machine code to construct control-flow graphs or to detect code-in-data. The instrumented version of the code is placed in a separate memory region (called the "clone"), so the original program code is not touched and as such neither data-in-code nor the variable-length property of 80x86 instructions pose a problem.

The behaviour of *DIOTA* can be influenced by using so-called backends. These are dynamic libraries that link against *DIOTA* and tell it what kind of instrumentation should be performed. They can ask for any dynamically linked routine to be intercepted and replaced with a routine of their own, ask to be notified of each memory access, of each basic block that is executed and of each system call that is performed (both before and after their execution, so their behaviour can be modified as well as analysed).

4.2 Description of the Data Race Backend of *DIOTA*

The data race detection backend [7] is implemented as a shared library that uses the services provided by *DIOTA*. The backend requests *DIOTA* to instrument all memory operations and to intercept all `pthread` synchronisation operations and provides functions that should be called whenever a memory or synchronisation operations occurs. The data race detector works as follows:

– For all parallel pieces of code, the memory operations are collected and compared. This is based on the fact that all memory operations between

[2] A dynamically linked helper program is used to instrument statically linked binaries.

two successive synchronisation operations (called segments) satisfy the same concurrency relation: they are either all parallel or not parallel with a given operation and therefore with the segment containing the latter operation. Given the sets $L(i)$ and $S(i)$ containing the addresses used by the load and store operations in segment i, the parallel segments i and j contain racing operations if and only if

$$\Big(\big(L(i) \cup S(i)\big) \cap S(j)\Big) \cup \Big(\big(L(j) \cup S(j)\big) \cap S(i)\Big) \neq \emptyset$$

Therefore, data race detection basically boils down to collecting the sets $L(i)$ and $S(i)$ for all segments executed and comparing parallel segments.

– In order to detect parallel segments, a vector timestamp [8, 9] is attached to each segment. As vector clocks are able to represent the *happened-before* [10] relation (they are strongly consistent), two vector clocks that are not ordered must belong to parallel segments. This gives us an easy way to detect parallel segments.

4.3 Modifications to Support Data Race Detection in Signal Handlers

In order to detect data races in sequential programs, the backend was adapted as follows (the numbers refer to the annotated Figure 4):

❶ the data race detector is informed when the application installs a signal handler. The current segment is ended, a new segment is started and the vector timestamp is saved (as `VC_install`).

❷ the data race detector is informed when the application sends a signal. If the target of the signal is the application itself, the current segment is ended, a new segment is started and the vector timestamp is saved (as `VC_signal`).

❸ each time signals become unblocked (by exiting a signal handler, or by using `sigprocmask`), a new segment is started and the vector timestamp is saved (as `VC_install`) for each signal that becomes unblocked.

❹,❺ the data race detector is informed at the begin and at the end of the execution of a signal handler. The backend then uses a new thread number for all memory and synchronisation operations executed by this signal handler. The initial vector timestamp for the signal handler is

- if the signal was sent by the application itself (can be detected by checking the signal context), the signal handler gets the vector clock that was saved when the signal was sent (`VC_install`).
- if the signal was sent by another application, the signal handler gets the vector clock that was saved when the signal handler was registered with the kernel (`VC_signal`).

The additional space required is therefore the space needed to store two vector timestamps for each signal handler (there are 64 possible signals in Linux). The actual size of such a vector timestamp can become quite large as each execution

of a signal handler gets a new thread number. The data race backend can cope with arbitrarily large vectors as as they are enlarged when a new thread starts. The potentially large memory consumption could be limited by using *accordeon clocks* [11] that can shrink as threads exit or by re-using thread numbers.

5 Experimental Evaluation

In order to test our implementation, we used a number of widely used Linux applications. Using the data race backend is easy: after compiling *DIOTA*, type `diota-dr` and the data race detector will attach itself to applications you start in the same shell. As the data race detector intercepts all memory operations, a huge slowdown should be expected (e.g. the mozilla browser incurs a slowdown of 63.4× [12]).

Our test revealed a number of applications with data races:

vim: (an editor) resizing the VIM window results in a `SIGWINCH` handler setting the variable `do_resize` to `TRUE`. This variable is then checked in the main event loop of VIM, where appropriate action is taken. Although our data race backend flags this as a data race, this is actually no real data race. This technique (setting a boolean in the signal handler and dealing with the signal in the main loop) is used in a lot of applications (e.g. the pine e-mail client, the Apache web server,...). The reason is that, although the signal handler is executed on behalf of the application, a signal handler is restricted in its capability to execute kernel calls. Delaying the actual signal handling so that the main program can handle it at a later point helps overcome this problem.

links: (a text-only web browser): resizing the window or pressing `^C` causes `queue_event()` to enter this event in a global queue (`ditrm`). As such, handling queued events while resizing causes havoc.

lynx: (another text-only web browser): global variables `LYlines` and `LYcols` are used for the width and height of the window. The `SIGWINCH` handler changes these variables. The `highlight()` function highlights a link and uses `LYcols` to check the available space.

Although we only tested a small number of applications, we were surprised that we found data races in signal handlers, which clearly shows that developers don't pay much attention to this kind of problem.

6 Conclusion

In this paper, we have shown that data races can show up in sequential programs. Fortunately, extending data race detectors for sequential programs is fairly straightforward: use a temporary thread number during the execution of an asynchronous signal handler and use additional rules for updating the vector clocks. The experimental evaluation revealed a number of data races in widely used programs. *DIOTA* is released to the public under the GPL and can be downloaded from `http://www.elis.UGent.be/diota/`.

Acknowledgements

Jonas Maebe is funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). The research was funded by Ghent University, and by the Fund for Scientific Research-Flanders (FWO-Vlaanderen).

References

1. Netzer, R.H., Miller, B.P.: What are race conditions? some issues and formalizations. ACM Letters on Programming Languages and Systems (1992)
2. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems **15** (1997) 391–411
3. Ronsse, M., De Bosschere, K.: An on-the-fly Data Race Detector for RECPLAY, a Record/Replay System for Parallel Programs. In: Proceedings of the 16th ACM Symposium on Operating Systems Principles, Saint-Malo (1997) (on CD)
4. Stevens, W.R.: Advanced Programming in the UNIX Environment. Addison Wesley (1993)
5. Zalewski, M.: Delivering signals for fun and profit. http://razor.bindview.com/publish/papers/signals.txt (2001) RAZOR, BindView Corporation.
6. Maebe, J., Ronsse, M., De Bosschere, K.: DIOTA: Dynamic instrumentation, optimization and transformation of applications. In Charney, M., Kaeli, D., eds.: Compendium of Workshops and Tutorials Held in conjunction with PACT'02: Intl. Conference on Parallel Architectures and Compilation Techniques, Charlottesville, VA (2002)
7. Ronsse, M., De Bosschere, K.: Non-intrusive detection of synchronization errors using execution replay. Automated Software Engineering **9** (2002) 95–121
8. Mattern, F.: Virtual time and global states of distributed systems. In Cosnard, Quinton, Raynal, Roberts, eds.: Proceedings of the Intl. Workshop on Parallel and Distributed Algorithms. Elsevier Science Publishers B.V., North-Holland (1989) 215–226
9. Fidge, C.J.: Logical time in distributed computing systems. In: IEEE Computer. Volume 24. (1991) 28–33
10. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21** (1978) 558–565
11. Christiaens, M., De Bosschere, K.: Accordion clocks: Logical clocks for data race detection. In Sakellariou, R., Gurd, J., Freeman, L., eds.: Proceedings of the 7th International Euro-Par Conference, Manchester, Springer (2001) 494–503
12. Ronsse, M., Stougie, B., Maebe, J., De Bosschere, K.: An efficient data race detector backend for diota. In: Proceedings of the International Conference ParCo2003. (2004) To be published.