Visualizing Data Races in Concurrent Signal Handlers[*]

Lin Gan, Guy Martin Tchamgoue, and Yong-Kee Jun[**]

Department of Informatics, Gyeongsang National University,
Jinju 660-701, South Korea
`ganlin428@gmail.com,guymt@ymail.com,jun@gnu.ac.kr`

Abstract. Asynchronous signal handling introduces fine-grained concurrency into sequential programs making them prone to data races. Unfortunately, existing tools for detecting data races in sequential programs that use concurrent signal handlers fail to provide effective means for understanding the dynamic behavior of concurrent signal handlers involved in data races. Thus, this paper presents a visualization tool that uses vertically parallel arrows to capture the logical concurrency between a sequential program and its concurrent signal handlers, materializes synchronization patterns with horizontal arrows, and uses colored squares to represent accesses to shared variables in order to provide a partial ordering of events that occured at runtime.

Key words: Data races, sequential programs, concurrent signal handlers, visualization

1 Introduction

Data races [2] represent one of the most notorious class of concurrency bugs in shared memory parallel programs. Data races occur when two threads access a shared memory location without proper synchronization, and at least one of the accesses is a write. Sequential programs are prone to data races due to asynchronous signals that introduce fine-grained concurrency into such programs making difficult to be thoroughly tested and debugged. Many tools [3–6] have recently been proposed to automatically detect data races in sequential programs that use concurrent signal handlers.

Ronsse et al. [3] adapted an existing on-the-fly race detector for multithreaded programs to fit for sequential programs. Tahara et al. [4] presented an approach for race detection in sequential programs that use signals. The technique is based

[*] This research was supported by the MKE (The Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program (NIPA-2012-H0301-12-3003) supervised by the NIPA (National IT Industry Promotion Agency).

[**] Corresponding author: In Gyeongsang National University, he is now the director of GNU Embedded Software Center for Avionics (GESCA) which is a national IT Research Center (ITRC) of Republic of Korea

on the */proc* system file (for Solaris 10) or the debug registers (for IA32 Linux) and uses watchpoints to monitor accesses to shared variables. Tchamgoue et al. [5, 6] proposed an efficient on-the-fly data race detection technique for sequential programs with concurrent signal handlers by using a lightweight labeling scheme to generate concurrency information with constant size for the sequential program and every instance of the concurrent signal handlers.

Despite the capabilities of all these tools, their outputs still consist of overly concise reports, or very long program traces [1]. Hence, understanding the runtime behavior of a sequential program together with its concurrent signal handlers and the reported data races is still difficult.

2 Visualization Model

This paper presents a visualization tool that uses vertically parallel arrows to capture the logical concurrency between the sequential program and its concurrent signal handlers, materializes synchronization patterns with horizontal arrows, and uses colored squares to represent accesses to shared variables in order to provide a partial ordering of events that occured at runtime. Our visualization model is therefore based on an easy to access partial order execution graph that provides programmers with information on reported data races and the runtime behavior of a program at a glance.

To understand the inner behavior of a sequential program with concurrent signal handlers and to understand the reported data races, a powerful visualization tool should be able to abstract even complex events for an easy visualization. To visualize data races in sequential programs, we should be able to efficiently represent accesses to shared variables and other concurrency patterns like signal registration, signal blocking and signal blocking.

As shown in Fig.1, we use letters R and W into a small squared shapes to respectively represent read and write accesses to shared variables. The instance number for each invocation is shown on top of the access. A bidirectional arrow is used for all signal registration operations. Similarly, unidirectional arrows are used for blocking and unblocking operations on signals. However, to differentiate between the main program or a concurrent signal that initiates an operation, a dot marker is used. Each arrow is topped with a code (i.e. `Rg` for registration, `Bk` for blocking, and `Ub` for unblocking) identifying each operation, followed by the signal number (e.g. `Rg:SIGALRM` in Fig.1 for the registration of the `SIGALRM` signal).

Vertically parallel arrows are used to represent the logical concurrency between the main program and its concurrent signal handlers as presented in the example of Fig.2. We abstract all invocations of the same signal handler into one vertical arrow. Thus, the signal handler for `SIGALRM` invoked two times as shown in Fig.2, is only materialized by a single line. Thus, the maximum number of vertical arrows in a visualization instance is always equal to the number of signals in the system plus one, i.e 65 for a UNIX-like system that maintains only 64 signals. However, each access to a shared variable is distinguished by

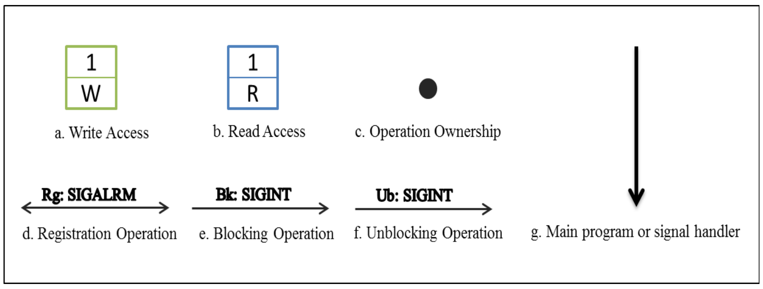

Fig. 1. Visualization Patterns

the invocation number of the signal handler to which it belongs. This number is always one for the main program as it is invoked only once. It is therefore easy to see that the main program registered two signal handlers: `SIGALRM` and `SIGINT`. The bidirectional arrow for the registration operation can be traversed in one way or another. Right after the registrations, `SIGALRM` was invoked for the first time and performed a write access to a shared variable. After this, `SIGINT` is invoked to block `SIGALRM`, perform a write operation on the shared variable and finally unblock `SIGALRM` it previously blocked. Similarly, the main program blocks `SIGINT`, performs a read access on the shared variable before unblocking `SIGINT`. After these operations, Fig.2 shows that the main program performs a write access on the same shared variable. We note that only accesses to a selected shared variable are kept visible.

Following the example of Fig.2, it is clear that the underlying program contains two data races involving the two accesses on the main program and the write access from the second invocation of the `SIGALRM` signal handler. This is due to the fact that there is no path from the write access of the second invocation of `SIGALRM` to the accesses of the main program. However, we can always find a path from one of the other accesses to another, meaning they are ordered by the happens-before relation. This simple but powerful visualization model therefore capture the partial ordering of runtime events in a sequential program that use concurrent signal handlers.

3 Conclusion

Asynchronous signal handling introduce fine-grained concurrency into sequential programs making prone to data races and difficult to be effectively tested and debugged. Data races may lead programs into non-deterministic executions with unpredictable results. In this paper, we presented a simple but powerful visualization tool that capture the dynamic runtime behavior of a sequential program together with its concurrent signal handlers into an partial order execution graph. By visualizing the data races detected at runtime, this tool provides a great understanding of the program to programmers.

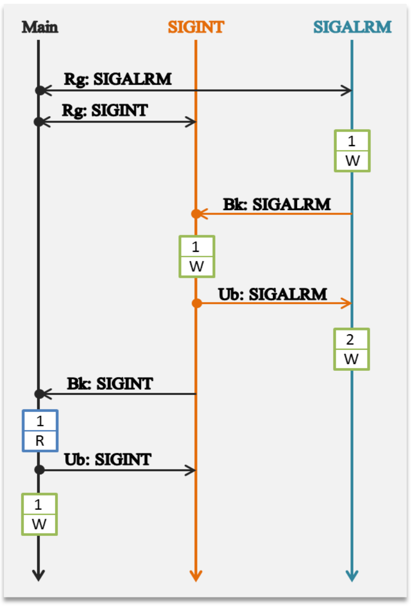

Fig. 2. Example of Program Visualization

However, the proposed visualization tool fails to represent re-entrant signal handlers that is signal handlers that can preempt themselves. We intend to extend this model to effectively represent not only re-entrant signal handlers, but also to support interrupt-based programs.

References

1. Artho, C., Havelund, K., Honiden, S.: Visualization of Concurrent Program Executions. In: The 31st Annual International Computer Software and Applications Conference (COMPSAC'07), pp.541–546. IEEE (July 2007)
2. Banerjee, U., Bliss, B., Ma, Z., Petersen, P.: A Theory of Data Race Detection. In: 4th Workshop on Parallel and Distributed programming: Testing, Analysis and Debugging (PADTAD'06), pp.69-78, ACM, (2006)
3. Ronsse, M., Maebe, J., and De Bosschere, K.: Detecting Data Races in Sequential Programs with DIOTA. In: Euro-Par 2004, LNCS, vol. 3149, pp.82–89, Springer, Heidelberg (2004).
4. Tahara, T., Gondow, K., and Ohsuga, S.: Dracula: Detector of Data Races in Signals Handlers. In: The 15th IEEE Asia-Pacific Software Engineering Conference (APSEC'08), pp.17–24, IEEE (2008).
5. Tchamgoue, G. M., Ha, O.-K., Kim, K.-H., and Jun, Y.-K.: Lightweight Labeling Scheme for On-the-fly Race Detection of Signal Handlers. In: Ubiquitous Computing and Multimedia Applications (UCMA'11), pp.201–208, Springer (2011).
6. Tchamgoue, G. M., Kim, K.-H., and Jun, Y.-K.: Efficient Detection of Data Races in Concurrent Signal Handlers. Information-An International Interdisciplinary Journal, 15(3):1317–1338, (March 2012).