Concurrent Breakpoints

Chang-Seo Park
EECS Department, UC Berkeley, USA
parkcs@cs.berkeley.edu

Koushik Sen
EECS Department, UC Berkeley, USA
ksen@cs.berkeley.edu

Abstract

In program debugging, reproducibility of bugs is a key requirement. Unfortunately, bugs in concurrent programs are notoriously difficult to reproduce because bugs due to concurrency happen under very specific thread schedules and the likelihood of taking such corner-case schedules during regular testing is very low. We propose concurrent breakpoints, a light-weight and programmatic way to make a concurrency bug reproducible. We describe a mechanism that helps to hit a concurrent breakpoint in a concurrent execution with high probability. We have implemented concurrent breakpoints as a light-weight library for Java and C/C++ programs. We have used the implementation to deterministically reproduce several known non-deterministic bugs in real-world concurrent Java and C/C++ programs with almost 100% probability.

1. Motivation

A key requirement in program debugging is reproducibility. Developers require that a bug can be reproduced deterministically so that they can confirm the bug and run the buggy execution repeatedly with the aid of a debugger and find the cause of the bug. For sequential programs, a bug can be reproduced deterministically by replaying the program with the recorded inputs and other sources of non-determinism. Bugs in sequential programs can be reported easily to a bug database because a user only needs to report the input on which the sequential program exhibits the bug.

Unfortunately, bugs in concurrent programs happen under very specific thread schedules and are often not reproducible during regular testing. Such non-deterministic bugs in concurrent programs are called *Heisenbugs*. One could argue that Heisenbugs could be made reproducible if the thread schedule is recorded along with program inputs during a program execution. Recording and replaying a thread schedule poses several problems: 1) It requires to observe the exact thread schedule either through program instrumentation or by using some specialized hardware. Instrumentation often incurs huge overhead and specialized hardware are often not readily available. 2) Replaying a thread schedule requires a special runtime which could again incur huge overhead.

Nevertheless, we need some information about the thread schedule along with the program inputs to reproduce a Heisenbug. We would like the information about thread schedules to be portable so that we do not need a special runtime to reproduce the bug. In this paper, we propose a simple light-weight technique called *concurrent breakpoints*, to specify enough information about a Heisenbug so that it can be reproduced with very high probability without requiring a special runtime or a full recording of the thread schedule.

Our technique for reproducibility is based on the observation that Heisenbugs can often be attributed to a *small* set of program states, called *conflict states*. A program execution is said to be in a conflict state if there exists two threads that are either 1) trying to access the same memory location and at least one of the accesses is a write (i.e. a data race), or 2) they are trying to operate on the same synchronization object (e.g. contending to acquire the same lock). Depending on how a conflict state is resolved, i.e. which thread is allowed to execute first, a concurrent program execution could end up in different states. Such difference in program states often lead to Heisenbugs. Therefore, in order to reproduce a Heisenbug, one should be able to reach those small set of conflict states and control the program execution from those states.

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA
Copyright © 2012 ACM 978-1-4503-1160-1/12/02...$10.00

2. Concurrent Breakpoints

In this paper, we propose *concurrent breakpoints*, a light-weight and programmatic tool that facilitates reproducibility of Heisenbugs in concurrent programs. A concurrent breakpoint is an object that defines a set of program states and a scheduling decision that the program needs to take if a state in the set is reached. Typically, the states described by a concurrent breakpoint would be a set of conflict states.

Formally, a *concurrent breakpoint* is a tuple (ℓ_1, ℓ_2, ϕ), where ℓ_1 and ℓ_2 are program locations and ϕ is a predicate over the program state. A program execution is said to have triggered a concurrent breakpoint (ℓ_1, ℓ_2, ϕ) if the following conditions are met

- the program reaches a state that satisfies the predicate: $\exists t_1, t_2 \in \mathit{Threads}.(t_1.pc = \ell_1) \wedge (t_2.pc = \ell_2) \wedge (t_1 \neq t_2) \wedge \phi$, and
- from the above state, the program executes the next instruction of thread t_1 before the next instruction of thread t_2.

That is, we say that a concurrent breakpoint (ℓ_1, ℓ_2, ϕ) is triggered if the program reaches some program state and takes an action from the state. The state is such that it satisfies the predicate ϕ and there exists two threads t_1 and t_2 such that t_1 and t_2 are at program locations ℓ_1 and ℓ_2 in the state, respectively. The action at the state executes the thread t_1 before the thread t_2.

Note that in our definition, a concurrent breakpoint involves two threads. The definition can be easily extended to involve more than two threads. For example, a concurrent breakpoint $(\ell_1, \ell_2, \ell_3, \phi)$ involves three threads.

Example. We can trigger a feasible data race in a program, i.e. reach a state in which two threads are about to access the same memory location and at least one of them is a write, using a concurrent breakpoint as follows. Consider the program in Figure 1. The concurrent breakpoint $(2, 5, t_1.\texttt{p1} == t_2.\texttt{p2})$ represents the state where two threads are at lines 2 and 5, respectively, and are about to access the same memory location denoted by the field `x` of the object referenced by both `p1` and `p2` and at least one of the accesses is a write. (If v is a local variable of thread t, then we denote the variable using $t.v$.) Such a racy state described by the concurrent breakpoint could be reached if `foo` and `bar` are executed in parallel by different threads on the same `Point` object. The concurrent breakpoint also specifies that if the racy state is ever reached, then the thread reaching line number 2 must execute its next instruction before the thread reaching line number 5 executes its next instruction. This forces the program to resolve the data race in a particular order. The commented out code represents the concurrent breakpoint in terms of library function calls. Inter-

```
void foo (Point p1) {
   // (new ConflictBreakpoint("bp11",p1)).
        breakHere(false,Global.TIMEOUT);
   p1.x = 10;
}
void bar (Point p2) {
   // (new ConflictBreakpoint("bp1",p2)).
        breakHere(true,Global.TIMEOUT);
   t = p2.x;
}
```

Figure 1. Example with data race

ested readers are referred to our technical report [4] for full details of the library and its implementation.

Concurrent breakpoints could represent all conflict states, i.e. they could represent data races and lock contentions. Concurrent breakpoints could represent other buggy states, such as a deadlock state or a state where an atomicity violation or a missed notification happens. We argue in [4] that the necessary information about a buggy schedule could be represented using a small set of concurrent breakpoints: if a program execution could be forced to reach all the concurrent breakpoints in the set, then the execution hits the Heisenbug.

Given a concurrent breakpoint (ℓ_1, ℓ_2, ϕ), it is very unlikely that two threads will reach statements labelled ℓ_1 and ℓ_2, respectively, at the same time in a concurrent execution, even though each thread could reach the statements independent of the other threads several times during the execution. Therefore, a concurrent breakpoint could be difficult to hit during a normal concurrent execution.

We describe a mechanism called BTRIGGER that tries to force a program execution to a concurrent breakpoint. We rewrite the predicate for a concurrent breakpoint as: $\exists t_1, t_2 \in \mathit{Threads}.\ (t_1 \neq t_2) \wedge \phi_{t_1} \wedge \phi_{t_2} \wedge \phi_{t_1 t_2}$, where ϕ_{t_1} only refers to local variables of thread t_1, ϕ_{t_2} only refers to local variables of thread t_2, and $\phi_{t_1 t_2}$ refers to local variables of both t_1 and t_2. BTRIGGER works as follows. During the execution of a program, whenever a thread reaches a state satisfying the predicate ϕ_{t_i} where $i \in \{1, 2\}$, we postpone the execution of the thread for T time units and keep the thread in a set $\mathtt{Postponed}_{t_i}$ for the postponed period. We continue the execution of the other threads. If another thread, say t, reaches a state satisfying the predicate ϕ_{t_j} where $j \in \{1, 2\}$ then we do the following. If there is a postponed thread, say t', in the set $\mathtt{Postponed}_{t_i}$ where $i \neq j$ and local states of the two threads t and t' satisfy the predicate $\phi_{t_1 t_2}$, then we report that the concurrent breakpoint has been reached. Otherwise, we postpone the execution of the thread t by T time units and keep the thread in the set $\mathtt{Postponed}_{t_j}$ for the postponed period. If the concurrent breakpoint is reached, we also order the execution of threads t and t' according to the order given by the concurrent breakpoint. Note that we do not postpone the execution of a thread indefinitely because this could result in a deadlock if all threads reach either ℓ_1 or ℓ_2 and none of the breakpoint predicates are satisfied by any pair of postponed threads.

BTRIGGER ensures that if a thread reaches a state satisfying the concurrent breakpoint partially (i.e. reaches a state satisfying the predicate ϕ_{t_1} or ϕ_{t_2}), it is paused for a reasonable amount of time, giving a chance to other threads to catch up and create a state that completely satisfies the concurrent breakpoint. This simple mechanism significantly increases the likelihood of hitting a concurrent breakpoint.

Our idea about concurrent breakpoints is motivated by recent testing techniques for concurrent programs, such as CalFuzzer [3], CTrigger [5], and PCT [1]. IMUnit [2], which is also closely related to this work, proposes a novel language to specify and execute schedules for multithreaded tests based on temporal logic and instrumentation of code.

Benchmark	LoC[1]	Runtime (seconds)			Breakpoint type	Prob.[3]
		Normal	w/ ctr	OVH[2]		
cache4j	3897	1.992	2.089	4.9	race	1.00
			2.051	3.0	atomicity	1.00
hedc	30K	1.780	3.835	115.4	race1	1.00
log4j 1.2.13	32K	0.190	0.208	9	deadlock	1.00
		0.135	-	-	missed-notify	1.00
logging	4250	0.140	0.140	0	deadlock	1.00
lucene	171K	0.136	0.159	17	deadlock	1.00
moldyn	1290	1.098	1.204	9.7	race	1.00
montecarlo	3560	1.841	2.162	17.4	race	1.00
raytracer	1860	1.097	1.274	16.1	race	1.00
stringbuffer	1320	0.131	0.159	21	atomicity	1.00
swing	422K	0.902	12.003	1230	deadlock	0.99
syncList	7913	0.134	0.142	6	atomicity	1.00
		0.131	0.134	2	deadlock	1.00

Table 1. Experimental results for Java programs ([1]Lines of code. [2]Overhead(%). [3]Empirical probability of triggering bug.)

3. Results

We have implemented concurrent breakpoints and BTRIGGER as a light-weight library (containing a few hundreds of lines code) for Java and C/C++ programs. We have used the implementation to reproduce several known Heisenbugs in real-world Java and C/C++ programs involving 1.6M lines of code. The breakpoints are inserted as extra code in the program under test using bug reports produced by CalFuzzer [3] and actual bug reports in bug databases. We ran each program with the breakpoints 100 times to measure the empirical probability of hitting the breakpoint. In our experiments, concurrent breakpoints made these non-deterministic bugs almost 100% reproducible.

Table 1 summarizes the results for our experiments on Java programs. For most of the benchmarks, the overhead of running the program with the concurrent breakpoint library was within 40% of the normal runtime. However, in some cases where we increased the waiting time to achieve a higher probability of hitting the breakpoint, the overhead became as large as 13x. Interested readers are referred to our technical report [4] for more detail on the results, including the results for C/C++ programs, and the methodology for inserting concurrent breakpoints.

4. Conclusion

Traditionally, programmers have used various ad-hoc tricks, such as inserting sleep statements and spawning a huge number of threads, to make a Heisenbug reproducible. These tricks are often found in various bug reports that are filed in open bug databases. We proposed a more scientific and programmatic technique to make a Heisenbug reproducible.

Acknowledgements
Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227), by NSF Grants CCF-101781, CCF-0747390, CCF-1018729, and CCF-1018730, and by a DoD NDSEG Graduate Fellowship. The last author is supported in part by a Sloan Foundation Fellowship. Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung.

References

[1] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS '10*, pages 167–178, New York, NY, USA, 2010. ACM.

[2] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov. Improved multithreaded unit testing. In *ESEC/FSE '11*, pages 223–233, New York, NY, USA, 2011. ACM.

[3] P. Joshi, M. Naik, C.-S. Park, and K. Sen. CalFuzzer: An extensible active testing framework for concurrent programs. In *CAV '09*, pages 675–681, Berlin, Heidelberg, 2009. Springer-Verlag.

[4] C.-S. Park and K. Sen. Concurrent triggers. Technical Report UCB/EECS-2011-156, EECS Department, UC Berkeley, Dec 2011.

[5] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS*, pages 25–36. ACM, 2009.