UPC-THRILLE Demo

Chang-Seo Park
Correctness Group Highlights

- **Active Testing** for Java, C, and UPC
  - Practical, easy-to-use tools for finding bugs, with sophisticated program analysis internally

- **Lightweight Specs for Parallelism**
  - By focusing just on parallelism, can we develop simple specifications that greatly improve our ability to find real parallelism bugs?

- **Concurrit DSL** for testing parallel code
Lightweight Parallel Specs

- **Goal:** Lightweight specifications for parallelism correctness.
  - Easy for programmers to write
  - Greatly increase effectiveness in testing, debugging, and verifying parallel programs

- **Semantic determinism**
  [FSE’09 (best paper), CACM’10, ICSE’10 (IFIP TC2 Manfred Paul)].

- **Semantic atomicity** [ASPLOS’11].

- **Nondeterministic sequential specs for parallel correctness** [HotPar’10, PLDI’11, PPoPP’12].
Key: Decompose effort in addressing parallelism and functional correctness.
Goal: Localize bug in a SEJITS execution.
Correctness Group Highlights

- **Active Testing** for Java, C, and UPC
  - Practical, easy-to-use tools for finding bugs, with sophisticated program analysis internally

- **Lightweight Specs for Parallelism**
  - Easy to write and, with testing, effective in finding real parallelism bugs
  - Determinism, atomicity, and NDSeq

- **Concurrit DSL for Testing Parallel Code**
  - Can we combine programmer intuition with testing techniques to find, reproduce bugs?
Concurrit: Domain Specific Language for Writing Concurrent Tests

Insights/ideas about thread schedules

Systematically explore all-and-only thread schedules specified by DSL

Specify a set of schedules in **formal, concise, and convenient** way
Concurrirt Demo

Tayfun Elmas
Simple Test for an Apache bug

Now suppose there are 3 threads, A, B, C running `testfunc`.
Threads A and B call `js_DestroyContext` and thread C calls `js_NewContext`.
First thread A removes its context from the runtime list. That context is not
the last one so thread does not touch `rt->` state and eventually calls `js_GC`.
The latter skips the above check and tries to to take the GC lock.
Before this moment the thread B takes the lock, removes its context from the
runtime list, discovers that it is the last, sets `rt->` state to LANDING, runs
the last-context-clean-up, runs the GC and then sets `rt->` state to DOWN.
At this stage the thread A gets the GC lock, setup itself as the thread that
runs the GC and releases the GC lock to proceed with the GC
when `rt->` state is DOWN.
Now thread C enters the picture. It discovers under the GC lock in
`js_NewContext` that the newly allocated context is the first one. Since
`rt->` state is DOWN, it releases the GC lock and starts the first context
initialization procedure. That procedure includes the allocation of the initial
atoms and it will happen when the thread A runs the GC.
This may lead precisely to the first stack trace from the comment 4.

---

**Figure 2.** Bug scenario, taken from Comment #5 of the bug report,
describing an interleaving of threads for the program in Figure 1.

```c

ExactScheduleTest:
1   Tid tA, tB, tC = WaitForDistinctThreads(3, EntersFunc(JS_NewContext));
2   RunThreadsUntil(tA, tB, EntersFunc(JS_DestroyContext));
3   RunThreadUntil(tA, InFunc(js_GC) && ReadsMem(&rt-> state));
4   RunThreadUntil(tB, ThreadEnds);
5   RunThreadUntil(tA, InFunc(js_GC) && WritesMem(&rt-> gcNumber));
6   RunThreadUntil(tC, EntersFunc(js_AddRoot));
7   RunThreadUntil(tA, ReturnsFunc(js_GC)); // violates assertion!
```
Concurrit

- **Implementation**: DSL embedded in C++

- Can write tests for
  - **Unit testing**:
    - Both manual and automated (Pin) instrumentation
  - **System testing**:
    - Manual instrumentation (lightweight and portable)
    - Test servers, e.g. Memcached, MySQL, Apache Httpd.
Example: Producer/consumer

Bounded buffer

Produce 4 items
Produce 4 items

C1
C2

Error:
Consumer reads from uninitialized cell.
How to reproduce a concurrency error? (Consumer reads from uninitialized cell.)

- Run 1000 times:
  - No guarantee
- Insert sleeps:
  - Useful but ad hoc, informal
- Concurrit approach
  - Write test to search for buggy schedules
Concurrit: Domain Specific Language for Writing Concurrent Tests

Insights/ideas about thread schedules

Systematically explore all-and-only thread schedules specified by DSL

Specify a set of schedules in formal, concise, and convenient way
TESTCASE() {

TVAR(P1); TVAR(P2);
TVAR(C1); TVAR(C2);

WAIT_FOR_DISTINCT_THREADS(
  (P1, P2), ENTERS(producer_routine));

WAIT_FOR_DISTINCT_THREADS(
  (C1, C2), ENTERS(consumer_routine));

WHILE (!ALL_ENDED(P1, P2, C1, C2)) {
  TVAR(t);

  CHOOSE_THREAD_BACKTRACK( 
    t, (P1, P2, C1, C2));

  RUN_THREAD_THROUGH(
    t, READS() || WRITES() || CALLS() 
    || ENTERS() || RETURNS());

}
}
SearchAll: Search all schedules

```c
TESTCASE() {
    TVAR(P1);  TVAR(P2);
    TVAR(C1);  TVAR(C2);

    WAIT_FOR_DISTINCT_THREADS(
        (P1, P2), ENTERS(producer_routine));

    WAIT_FOR_DISTINCT_THREADS(
        (C1, C2), ENTERS(consumer_routine));

    WHILE (!ALL_ENDED(P1, P2, C1, C2)) {
        TVAR(t);

        CHOOSE_THREAD_BACKTRACK(
            t, (P1, P2, C1, C2));

        RUN_THREAD_THROUGH(
            t, READS() || WRITES() || CALLS() || ENTERS() || RETURNS());
    }
}
```

Instrumented to control Software Under Test (SUT)

Run captured threads

Software Under Test (SUT)

- **P1**: Produce 4 items
- **P2**: Produce 4 items
- **C1**: Consume 4 items
- **C2**: Consume 4 items
**SearchInFunc:** Localize search to particular functions and operations

```c
TESTCASE() {
    TVAR(P1);  TVAR(P2);
    TVAR(C1);  TVAR(C2);

    WAIT_FOR_DISTINCT_THREADS(
        (P1, P2), ENTERS(bounded_buf_put));

    WAIT_FOR_DISTINCT_THREADS(
        (C1, C2), ENTERS(bounded_buf_get));

    WHILE (!ALL_ENDED(P1, P2, C1, C2)) {
        TVAR(t);

        CHOOSE_THREAD_BACKTRACK(
            t, (P1, P2, C1, C2));

        RUN_THREAD_THROUGH(
            t, ENTERS() || RETURNS() || HITS_MANUAL_PC());
    }
}
```
TESTCASE() {
  TVAR(P1);
  TVAR(C1);  TVAR(C2);

  WAIT_FOR_THREAD(
    P1, ENTERS(bounded_buf_put));

  WAIT_FOR_DISTINCT_THREADS(
    (C1, C2), ENTERS(bounded_buf_get));

  RUN_THREAD_THROUGH(
    P1, RETURNS(bounded_buf_put));

  RUN_THREAD_THROUGH(
    C1, HITS_MANUAL_PC(42));

  RUN_THREAD_THROUGH(
    C2, RETURNS(bounded_buf_get));

  RUN_THREAD_THROUGH(C1, ENDS()); // ERROR!
}

P1
Insert item to buffer

C1
Check item and prepare to read

C2
Read item and update head
Read from new head (uninitialized slot)
Where We Ended Up

- **Active Testing** for Java, C, and UPC
  - Practical, easy-to-use tools for finding bugs, with sophisticated program analysis internally

- **Lightweight Specs for Parallelism**
  - Easy to write and, with testing, effective in finding real parallelism bugs
  - Determinism, atomicity, and NDSeq

- **Concurrit** DSL for Testing Parallel Code
  - Combine programmer intuition with automated testing techniques to find, reproduce bugs.