Detecting Concurrency Bugs using Static and Dynamic Program Analyses

Microsoft Research Asia

Verified Software Workshop and Summer School 2012

Aarti Gupta
NEC Labs America

Workshop: August 23–24, 2012
Summer School: August 27–31, 2012

Venue
East China Normal University
Shanghai, China

www.nec-labs.com
Detecting Concurrency Bugs using Static and Dynamic Program Analyses

Aarti Gupta
Systems Analysis & Verification Group
NEC Labs America

Acknowledgements:
Malay Ganai, Vineet Kahlon, Nadia Papakonstantinou, Chao Wang*

www.nec-labs.com
Motivation

Key Computing Trends

- Mobile
- Server
- Gaming

Low Power, High Performance

- Multi-core platforms everywhere
- Need parallel, multi-threaded programming

Parallel/Multi-threaded Programming

- Difficult to program
  - Dependencies due to shared data
  - Subtle effects of synchronizations
- Difficult to debug
  - too many interleavings of threads
  - hard to reproduce bugs

Data centers, Cloud platforms

- Distributed systems

Therac-25 medical radiation device (1985) malfunction due to SW race, at least 5 deaths
2003 Northeast Blackout Cost: $4 billion

Nasdaq's Facebook glitch came from 'race conditions'
Nasdaq may pay out as much as $13 million due to a hard-to-find software bug
Outline

- Motivation & Background
  1. Static Analysis
  2. Dynamic Analysis
  3. Coverage-guided Testing

- Conclusions
Common Concurrency Bugs

- **Race Condition**: simultaneous memory access (at least one write)

```c
/*--- Thread 1 ----*/
.
.
Write (globalVar);
.
.
/*--- Thread 2 ----*/
.
.
Read (globalVar);
.
.
```

- **Deadlock**: hold-and-wait cycles

```c
/*--- Thread 1 ---*/
lock(A);
.
.
lock(B);

/*--- Thread 2 ---*/
lock(B);
.
.
lock(A);
```

- **Atomicity violation**: interference from other threads/processes

```c
/*--- Thread 1 ----*/
if (account_ptr != NULL) {
    ...
    account_ptr -> amount -= debit;
}

/*--- Thread 2 ----*/
if (account_ptr != NULL) {
    free(account_ptr);
    account_ptr = NULL;
}
```
Data Race Bugs

- Two memory accesses conflict if
  - They target the same location
  - They are not both read operations

- Data Race: If two conflicting memory accesses happen concurrently

- Data races may reveal synchronization errors
  - Typically caused because programmer forgot to take a lock
  - Many programmers tolerate “benign” races
  - The C++0x/11 standard expects code to be free of data races
Data Race Detection

- **Two popular approaches for data race detection**
  - **Lockset analysis** [Savage et al. 97, ERASER]
    - *Lockset: set of locks held at a program location*
      - Compute locksets for all locations in a program (statically or dynamically)
      - *Race: When conflicting accesses from program locations with disjoint locksets*
    - Gives too many false warnings (locations may not be reachable concurrently)
  - **Happens-Before (HB) analysis** [Lamport 77]
    - *Happens-Before order: a partial order over synchronization events*
      - Observe HB order during dynamic execution
      - *Race: If conflicting accesses are not ordered by HB*
    - This is precise, but dynamic executions have limited coverage

- **Many variations and enhancements** [Naik et al., Flanagan et al. Rosu et al. …]
  - *Precision: how many false races does it report?*
  - *Coverage: how many real races does it cover?*
  - NEC’s CoBe framework: based on locksets with precise static analysis
void Alloc_Page ( ) {
    a = c;
    pt_lock(&plk);
    if (pg_count >= LIMIT) {
        pt_wait (&pg_lim, &plk);
        incr (pg_count);
        pt_unlock(&plk);
        sh1 = sh;
    } else {
        pt_lock (&count_lock);
        page = alloc_page();
        sh = 5;
        if (page)
            incr (pg_count);
        pt_unlock(&count_lock);
    } else {
        pt_lock (&count_lock);
        pt_unlock (&plk);
        page = alloc_page();
        sh = 5;
        if (page)
            incr (pg_count);
        pt_unlock(&count_lock);
    } else {
        pt_lock (&count_lock);
        pt_unlock (&plk);
        page = alloc_page();
        sh = 5;
        if (page)
            incr (pg_count);
        pt_unlock(&count_lock);
    }
    b = a+1;
}

void Dealloc_Page ( )
    pt_lock(&plk);
    if (pg_count == LIMIT) {
        sh = 2;
        decr (pg_count);
        sh1 = sh;
        pt_notify (&pg_lim, &plk);
    } else {
        pt_lock (&count_lock);
        pt_unlock (&plk);
        page = dealloc_page();
        sh = 5;
        if (page)
            incr (pg_count);
        pt_unlock(&count_lock);
        sh1 = sh;
        pt_notify (&pg_lim, &plk);
    }
}

Consider all possible pairs of locations where shared variables are accessed (e.g. for checking data races)
Motivating Example: Lockset Analysis

```c
void Alloc_Page ( ) {
    a = c;
    pt_lock(&plk);
    if (pg_count >= LIMIT) {
        pt_wait (&pg_lim, &plk);
        incr (pg_count);
        pt_unlock(&plk);
        sh1 = sh;
    } else {
        pt_lock (&count_lock);
        page = alloc_page();
        sh = 5;
        if (page)
            incr (pg_count);
        pt_unlock(&count_lock);
    }
}
```

```c
void Dealloc_Page ( )
    pt_lock(&plk);
    if (pg_count == LIMIT) {
        sh = 2;
        decr (pg_count);
        sh1 = sh;
        pt_notify (&pg_lim, &plk);
        pt_unlock(&plk);
    } else {
        pt_lock (&count_lock);
        page = alloc_page();
        sh = 5;
        if (page)
            incr (pg_count);
        pt_unlock(&count_lock);
    }
}
```

Lockset Analysis: Compute set of locks held at locations
Here, lock plk is held in both locations. Hence, these locations cannot be reached concurrently. Therefore, there is no data race.
void Alloc_Page ( ) {
    a = c;
    pt_lock(&plk);
    if (pg_count >= LIMIT) {
        pt_wait (&pg_lim, &plk);
        incr (pg_count);
        pt_unlock(&plk);
        sh1 = sh;
    } else {
        pt_lock (&count_lock);
        pt_unlock (&plk);
        page = alloc_page();
        sh = 5;
        if (page)
            incr (pg_count);
        pt_unlock(&count_lock);
    }
    end-if
    b = a+1;
}

void Dealloc_Page ( ) {
    pt_lock(&plk);
    if (pg_count == LIMIT) {
        sh = 2;
        decr (pg_count);
        sh1 = sh;
        pt_notify (&pg_lim, &plk);
        pt_unlock(&plk);
    } else {
        pt_lock (&count_lock);
        pt_unlock (&plk);
        page = alloc_page();
        sh = 4;
        incr (pg_count);
        pt_unlock(&count_lock);
    }
    end-if
}

Due to the wait-notify ordering constraint, these locations cannot be reached concurrently. Therefore, no data race.
void Alloc_Page ( ) {
    a = c;
    pt_lock(&plk);
    if (pg_count >= LIMIT) {
        pt_wait(&pg_lim, &plk);
        incr (pg_count);
        pt_unlock(&plk);
        sh1 = sh;
    } else {
        pt_lock (&count_lock);
        pt_unlock (&plk);
        page = alloc_page();
        sh = 5;
        if (page)
            incr (pg_count);
        pt_unlock(&count_lock);
    }
}

void Dealloc_Page ( )
    pt_lock(&plk);
    if (pg_count == LIMIT) {
        sh = 2;
        decr (pg_count);
        sh1 = sh;
        pt_notify (&pg_lim, &plk);
        pt_unlock(&plk);
    } else {
        pt_lock (&count_lock);
        pt_unlock (&plk);
        decr (pg_count);
        sh = 4;
        pt_unlock(&count_lock);
    }

How do we get these invariants?
By using abstract interpretation, model checking, ...

Data race?

NO, due to invariants at these locations
pg_count is in (-inf, LIMIT) in T1
pg_count is in [LIMIT, +inf) in T2
Therefore, these locations are not concurrently reachable
CoBe: Concurrency Bench Platform

- Fast front-end for generating warnings
- On-demand precision in back-end

Global Program Information
- Which data/objects are shared?
- What locks are held?
- Which code can run in parallel?

User Interface
- Fast Analysis (Front-end)
- Heavy-weight Analysis (Back-end)

CoBe Tool Viewer
- Warnings
  - Data races
  - Deadlocks
  - Serializability violations
- Result
  - Concrete error trace
  - No violation possible
  - Inconclusive (incomplete)
CoBe Highlights: Bootstrapping Pointer Analysis

- CoBe goals: Precision + Scalability
- Requirement: Flow-Sensitive Context-Sensitive (FSCS) alias analysis
  - For precisely identifying shared variables, locksets, concurrent reachability
  - Steensgaard’s pointer analysis is efficient, but flow- and context-insensitive
  - Observation: Need to track lock pointers and lock/unlock statements

```c
foo(){                     access_p(){                   access_r(){
  access_p();            lock(lock_ptr);               lock(lock_ptr);
  q = &b;                p = &a;                        r = &c;
  access_r();            unlock(lock_ptr);           unlock(lock_ptr);
  q = p;                 }                                }
  q = r;                 }                                }
```

- Key idea: Divide-and-conquer via Bootstrapping [Kahlon, PLDI 08]
  - Use Steensgaard’s analysis to derive initial partitions (equivalence classes)
  - Note: each pointer can only be aliased to pointers within its equivalence class
  - Carry out FSCS-alias analysis for refining the partitions
  - Scales well, as partitions of interest are usually small

- Example: initial partitions are `{lock_ptr, &lock_p, &lock_r}` , `{p, q, r, &a, &b, &c }`
CoBe: Experiments

- Linux device drivers with known data race bugs

<table>
<thead>
<tr>
<th>Linux Driver</th>
<th>KLOC</th>
<th>#Sh Vars</th>
<th>#Warnings</th>
<th>Time (sec)</th>
<th># After Invariants</th>
<th>Time (sec)</th>
<th>#Witness MC</th>
<th>#Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>pci_gart</td>
<td>0.6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>jfs_dmap</td>
<td>0.9</td>
<td>6</td>
<td>13</td>
<td>2</td>
<td>1</td>
<td>52</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>hugetlb</td>
<td>1.2</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ctrace</td>
<td>1.4</td>
<td>19</td>
<td>58</td>
<td>7</td>
<td>3</td>
<td>143</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>autosfs_expire</td>
<td>8.3</td>
<td>7</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>12</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>ptrace</td>
<td>15.4</td>
<td>3</td>
<td>1</td>
<td>15</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>raid</td>
<td>17.2</td>
<td>6</td>
<td>13</td>
<td>2</td>
<td>6</td>
<td>75</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>tty_io</td>
<td>17.8</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>11</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>ipoib_multicast</td>
<td>26.1</td>
<td>10</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>16</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>99</td>
<td></td>
<td>24</td>
<td></td>
<td>21</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>decoder</td>
<td>2.9</td>
<td>4</td>
<td>256</td>
<td>5min</td>
<td></td>
<td>15</td>
<td>22min</td>
<td></td>
</tr>
<tr>
<td>bzip2smp</td>
<td>6.4</td>
<td>25</td>
<td>15</td>
<td>18</td>
<td></td>
<td>12</td>
<td>35</td>
<td></td>
</tr>
</tbody>
</table>

After Phase 1 (Warning Generation)

After Phase 2 (Warning Reduction)

After Phase 3 (Model Checking)

August 2012
CoBe: Experiments

- **Linux device drivers with known data race bugs**

<table>
<thead>
<tr>
<th>Linux Driver</th>
<th>KLOC</th>
<th>#Sh Vars</th>
<th>#Warnings</th>
<th>Time (sec)</th>
<th># After Invariants</th>
<th>Time (sec)</th>
<th>#Witness MC</th>
<th>#Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>pci_gart</td>
<td>0.6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>jfs_dmap</td>
<td>0.9</td>
<td>6</td>
<td>13</td>
<td>2</td>
<td>1</td>
<td>52</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>hugetlb</td>
<td>1.2</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ctrace</td>
<td>1.4</td>
<td>19</td>
<td>58</td>
<td>7</td>
<td>3</td>
<td>143</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>autosps_expire</td>
<td>8.3</td>
<td>7</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>12</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>ptrace</td>
<td>15.4</td>
<td>3</td>
<td>1</td>
<td>15</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>raid</td>
<td>17.2</td>
<td>6</td>
<td>13</td>
<td>2</td>
<td>6</td>
<td>75</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>tty_io</td>
<td>17.8</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>11</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>ipoib_multicast</td>
<td>26.1</td>
<td>10</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>16</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td></td>
<td>99</td>
<td></td>
<td>24</td>
<td>21</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>decoder</td>
<td>2.9</td>
<td>4</td>
<td>256</td>
<td>5min</td>
<td>15</td>
<td>22min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bzip2smp</td>
<td>6.4</td>
<td>25</td>
<td>15</td>
<td>18</td>
<td>12</td>
<td>35</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Successfully applied to medium-sized Linux device drivers
- How about scalability on industry projects?
  - On large code (> 100 kLOC – 1 MLOC), could not create CFG for entry function
CoBe Highlights: Layered Analysis

- **Issue**
  - For threads executing functions with large CFGs, the CFG construction itself may run out of memory

- **Strategy**
  - Trade-off time for space, via Call Graph layering
  - Work with few layers in memory at a time, using files to transfer information between layers

- **Implementation**
  - First the CFG of the entry function in each thread is created up to some small depth cutoff (DC), e.g. DC = 2 includes main and foo above
  - The CFGs of functions called at depth greater than DC (e.g. bar, baz) are built on-the-fly, in depth-first order according to call graph of entry function (main)
  - Aliasing and lockset information is passed across layers.
int sh;

main(){
    foo(&sh);
}

foo(int *p){
    bar(p);
    baz(p);
}

bar(int *q){
    lock(lk);
    *q = 1;
    unlock(lk);
}

baz(int *r){
    *r = 0;
}

pass points-to set of p

pass points-to set of r

log the update to *q as a shared variable access

return empty lock set

log the update to *r as a shared variable access

return lock set {lk}
CoBe Application on NEC Projects

- **CoBe was applied on industry projects by collaborators in NEC Japan**
  
  Acknowledgement: Ohashi-san, Ikeda-san (NEC Japan)

<table>
<thead>
<tr>
<th>Application</th>
<th>KLOC</th>
<th>Warnings Reported</th>
<th>Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project – A</td>
<td>9</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Project – B</td>
<td>21</td>
<td>25</td>
<td>9</td>
</tr>
<tr>
<td>Project – C</td>
<td>26</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Project – D</td>
<td>40</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>Project – E</td>
<td>379</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

- All bugs were difficult to find by testing

**CoBe Summary**

- Technology highlights: Bootstrapping FSCS pointer analysis, layered flow, lock acquisition history analysis, context-dependent stubs for thread APIs, (optional) dataflow analysis
- Can routinely handle projects with large code base (10’s k – 1M LOC)
- Reported warnings are manually inspected, with debugging info from tool
Where we are so far …

1. Static Analysis
   - Pros
     • Automatic coverage over all inputs and all thread schedules
     • Reasonably efficient and scalable
   - Cons
     • Requires effort for manual inspection of reported warnings (may be false)
     • Difficult to apply in multi-process and distributed settings

☐ In the rest of the talk, focus on Dynamic Analysis and Testing
   • Can be applied more easily in multi-process and distributed settings
   • Based on actual executions, therefore no false warnings
   • However, need to amplify coverage (over thread schedules, over inputs)

2. Predictive Analysis
3. Coverage-guided Testing
Motivation

Concurrent program

Static Verification
e.g. model checking

Large state-space

Full static verification is often intractable

Alternate approaches

Concurrent program

Collect shared access footprint

Trace

Online/offline monitoring of trace

Predict errors in alternate interleavings

Monitoring problem
Tractable and no false alarms.

Will not talk about this

Predictive Analysis problem
Larger set of interleavings is explored.

Next

August 2012
Predictive Analysis (based on dynamic traces)

- **Predictive analysis** [Rosu et al. CAV 07, Farzan et al. TACAS 09, ...]
  - Step 1: Run a test execution and log information about events of interest
    - Synchronization operations, shared variable accesses
  - Step 2: Generate a *predictive model* over the observed events
    - Relax some ordering constraints, may lead to false bugs or missed bugs
  - Step 3: Analyze the predictive model to check alternate interleavings
    - Can use model checking or systematic testing etc.

- **Symbolic Predictive Analysis** [Wang et al. FM 09, TACAS 10]
  - Step 1: same as above
  - Step 2: Generate a *precise* predictive model by considering constraints due to synchronization *and* dataflow
    - No false bugs
  - Step 3: Symbolically explore all possible thread interleavings of events in that trace, *using an SMT solver*
    - Performs better than explicit enumeration
C program: multi-threaded, using Pthreads

Execution trace

Concurrent Trace Program (CTP)

“assume( c )” means the (c)-branch is taken
Symbolic Predictive Analysis using CTP

- Build an SMT formula (e.g. linear arithmetic)
  - $F_{\text{program}}$ : encodes all feasible thread interleavings of CTP
  - $F_{\text{property}}$ : encodes the property, e.g. an assertion violation, or an atomicity violation

- Solve using an SMT solver
  \[
  (F_{\text{program}} \wedge F_{\text{property}}) \wedge F_{\text{property}} \wedge F_{\text{program}}
  \]
  - Sat $\rightarrow$ found a real error
  - Unsat $\rightarrow$ no error in any interleaving

- Improves
  - Precision over other predictive techniques
  - Covers all possible interleavings of the observed events.
NEC BEST* Tool

* Binary-based Error-directed Symbolic Testing

Tool Highlights

- Based on Binaries
- Automatic bug scenarios
- No false bugs
- Easier for developers to use

[ Ganai et al. ASE 2011 ]

BEST Tool

Environment (Optional: Stubbed)

x86 Binary

Test Data

C/C++/
Java

Witness traces

HTML View

Socket/File Interface

gcc/g++/gcj

Binary based
Error directed
Symbolic Testing

Bin
ary
ased
Error
directed
Sym
bolic
Testing

Execute "Scheduled" Events

Record "Monitored" Events

Symbolic Predictor & Analyzer

Predict: Generate witnesses for inferred Concurrency violations

Analyze: Generate witness for observed "failure"

Observed trace

Bug Scenarios

Mismatch Communication

Improper Resource Utilization

Non-atomic Updates

Resource Contention

August 2012
Atomicity Violations

- **Atomicity is a desired correctness criterion**
  - Non-interference on accesses to shared variables
  - Serializability is a notion that checks atomicity

- A detailed study showed 69% of concurrency bugs are due to atomicity violations

[Lu et al. ASPLOS’08]
## BEST: Results for Atomicity Analysis

<table>
<thead>
<tr>
<th>Application (LOC, Lang)</th>
<th>Trace Information</th>
<th>Atomicity Inference</th>
<th>Causal Mutual Atomic</th>
<th>Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n-thr</td>
<td>#events</td>
<td>#lvars</td>
<td>#gvars</td>
</tr>
<tr>
<td>account (145, Java)</td>
<td>4</td>
<td>744</td>
<td>15</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>923</td>
<td>22</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>1239</td>
<td>32</td>
<td>67</td>
</tr>
<tr>
<td>tsp (713, Java)</td>
<td>7</td>
<td>31K</td>
<td>20</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>34K</td>
<td>30</td>
<td>47</td>
</tr>
<tr>
<td>aget (1.2K, C)</td>
<td>3</td>
<td>854</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>1005</td>
<td>8</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>1127</td>
<td>12</td>
<td>55</td>
</tr>
<tr>
<td>fastspy (1.5K, C)</td>
<td>11</td>
<td>1017</td>
<td>75</td>
<td>139</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>1972</td>
<td>85</td>
<td>203</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>2912</td>
<td>95</td>
<td>267</td>
</tr>
<tr>
<td>finalsolution (2K, C++)</td>
<td>56</td>
<td>2927</td>
<td>66</td>
<td>9857</td>
</tr>
<tr>
<td></td>
<td>67</td>
<td>3536</td>
<td>87</td>
<td>11870</td>
</tr>
<tr>
<td>prozilza (2.7K, C++)</td>
<td>8</td>
<td>8945</td>
<td>1035</td>
<td>9859</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>15K</td>
<td>1854</td>
<td>16075</td>
</tr>
<tr>
<td>axel (3.1K, C)</td>
<td>3</td>
<td>1452</td>
<td>204</td>
<td>1258</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>3016</td>
<td>244</td>
<td>2778</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>4872</td>
<td>284</td>
<td>4590</td>
</tr>
<tr>
<td>bzip2smp (6.4K, C)</td>
<td>4</td>
<td>2126</td>
<td>65</td>
<td>2185</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2153</td>
<td>61</td>
<td>2216</td>
</tr>
<tr>
<td>alsa (33K, C++)</td>
<td>5</td>
<td>14K</td>
<td>372</td>
<td>17K</td>
</tr>
</tbody>
</table>

- PIN is used to instrument pthread, and libc libraries dynamically at runtime
- YICES-1.0.28 SMT solver was used to find the violating traces.
Moving on to Testing …

User expectation:
If the program fails the given test, the user wants to see the bug

The reality:
Even if the program may fail (under a certain schedule), the user likely won’t see it

Why?
Thread scheduling is controlled by the OS and the Pthreads library

Tools: VeriSoft, Chess, Fusion, Inspect
• take control of the scheduler
• to execute alternate thread schedules
Testing Multi-threaded Programs

- **Common Practice**
  - Stress tests: creating many concurrent threads (with heavy workload and random delays) and run them repeatedly

- **Research Efforts**
  - Fuzz testing (by randomizing the thread schedules)
    - CalFuzzer (Sen et al., PLDI’08), CTrigger (Park et al., ASPLOS’09), PENELLOPE (Sorrentino et al., FSE’10), …
  - Model checking
    - SPIN, Java Pathfinder, …
  - Systematic concurrency testing
    - **DPOR**: dynamic partial order reduction [Flanagan & Godefroid, POPL 2005]
      - Sound reduction, but too many equivalence classes
    - **PCB**: preemptive context bounding [Musuvathi et al., OSDI 2008]
      - Unsound, still too many interleavings
    - **Fusion**: SMT-based search over all interleavings [Wang et al., FSE 09, FM09]
      - Sound, but symbolic search is still expensive

- **Question**: Which (subset of) interleavings to cover?
What is the root cause of a “concurrency bug”?  
- Programmers often make some implicit assumptions regarding the concurrency control of the program, but fail to enforce them!  
  - Certain blocks should be mutually exclusive → data race  
  - Certain blocks should be executed atomically → atomicity violation  
  - Certain operations should be executed in a fixed order → order violation  

To chase “concurrency bugs”, we would like to go after the “broken assumptions”…  
- Exhaustively test all concurrency control scenarios  
- But not all possible thread interleavings
Coverage-Guided Systematic Testing

[Wang et al. ICSE 2011]

- New Coverage Metric: HaPSet (History-aware Predecessor Set)

- We use this metric to guide testing
  - Strategy: Don’t test a thread interleaving if its “concurrency control scenario” (HaPSet) has already been covered
  - Use a systematic testing framework
    - e.g. stateless model checking, with dynamic partial order reduction (DPOR) or pre-emptive context bounding (PCB, as in CHESS)
  - Keep track of HaPSets covered by each explored execution
  - When backtracking, prune an interleaving if its HaPSet is already covered

- HaPSet is based on PSet (Predecessor Set)
PSet (Predecessor Set) [Yu & Narayanasamy ISCA 09]

PSet (statement): the set of immediately dependent "remote" statements

\[
\begin{align*}
\text{PSet}(W1) &= \{\} \\
\text{PSet}(R1) &= \{\} \\
\text{PSet}(R2) &= \{W1\} \\
\text{PSet}(R3) &= \{W1\} \\
\text{PSet}(R4) &= \{\} \\
\text{PSet}(W2) &= \{R3, R4\} \\
\text{PSet}(W3) &= \{W2\}
\end{align*}
\]

PSets are tracked for statements in code, not for events

PSets were used for enforcing safe executions at runtime, not for testing
1. Synchronization statements
   - PSet ignored synchronizations, e.g. lock/unlock, wait/notify
   - HaPSet considers synchronizations – essential for concurrency

2. Context & thread sensitivity
   - PSet (effectively) treats a statement as a (file, line) pair
   - HaPSet treats a “statement” as a tuple (file, line, thr, ctx), where
     - thr = \{local\_thread, remote\_thread\} (exploits symmetry)
     - ctx = truncated calling context (typically previous 5 levels)
Intuition: Why are HaPSets Useful?

From the given run
- \( \text{HaPSet}(e_1) = \{\} \)
- \( \text{HaPSet}(e_2) = \{e_1\} \)
- \( \text{HaPSet}(e_3) = \{\} \)
- \( \text{HaPSet}(e_4) = \{e_3\} \)

From all good runs
- \( \text{HaPSet}(e_1) = \{e_2\} \)
- \( \text{HaPSet}(e_2) = \{e_1, e_4\} \)
- \( \text{HaPSet}(e_3) = \{\} \)
- \( \text{HaPSet}(e_4) = \{e_3\} \)

Observations:
#1. In all good runs, \( \text{HaPSet}[e_3] = \{\} \)
#2. In all good runs, \( e_2 \) is not in \( \text{HaPSet}[e_4] \)

Need only 2 test runs to capture all “good” runs
How to use HaPSets for Guidance?

Thread T1

... 

{ }

if (p != 0)

e2

e3
*(p) = 10;

}

Thread T2

... 

{ 

p = &a;

}

{ 

p = 0;

}

Observations:

#1. In all good runs, HaPSet[e3] = { }

#2. In all good runs, e2 is not in HaPSet[e4]

Steer search directly to a “bad” run

From the given run

HaPSet(e1) = {}
HaPSet(e2) = {e1}
HaPSet(e3) = {}
HaPSet(e4) = {e3}

From all good runs

HaPSet(e1) = {e2}
HaPSet(e2) = {e1,e4}
HaPSet(e3) = {}
HaPSet(e4) = {e3}

From all (good and bad) runs

HaPSet(e1) = {e2}
HaPSet(e2) = {e1,e4}
HaPSet(e3) = {e4}
HaPSet(e4) = {e3,e2}
Thrift is a software framework by Facebook, for scalable cross-language services development.

The C++ library has **18.5K lines of C++ code**. It has a known **deadlock**.

<table>
<thead>
<tr>
<th>Test Program</th>
<th>LoC</th>
<th>thrds</th>
<th>bug type</th>
<th>HaPSet runs</th>
<th>time(s)</th>
<th>DPOR runs</th>
<th>time(s)</th>
<th>PCB0 runs</th>
<th>time(s)</th>
<th>PCB1 runs</th>
<th>time(s)</th>
<th>PCB2 runs</th>
<th>time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lib-w2-5t</td>
<td>18.5k</td>
<td>3</td>
<td>deadlk</td>
<td>14</td>
<td>27.8</td>
<td>23</td>
<td>18.6</td>
<td>512(no)</td>
<td>247.2</td>
<td>26</td>
<td>29.2</td>
<td>215</td>
<td>146.9</td>
</tr>
<tr>
<td>lib-w3-5t</td>
<td>18.5k</td>
<td>4</td>
<td>deadlk</td>
<td>18</td>
<td>27.5</td>
<td>733</td>
<td>TO</td>
<td>1301</td>
<td>TO</td>
<td>399</td>
<td>229.7</td>
<td>876</td>
<td>TO</td>
</tr>
<tr>
<td>lib-w4-5t</td>
<td>18.5k</td>
<td>5</td>
<td>deadlk</td>
<td>22</td>
<td>33.7</td>
<td>665</td>
<td>TO</td>
<td>1111</td>
<td>TO</td>
<td>980</td>
<td>TO</td>
<td>677</td>
<td>TO</td>
</tr>
<tr>
<td>lib-w5-5t</td>
<td>18.5k</td>
<td>6</td>
<td>deadlk</td>
<td>25</td>
<td>38.1</td>
<td>572</td>
<td>TO</td>
<td>899</td>
<td>TO</td>
<td>670</td>
<td>TO</td>
<td>582</td>
<td>TO</td>
</tr>
</tbody>
</table>
## Benchmarks: Extracted Mozilla/MySQL Bugs

<table>
<thead>
<tr>
<th>Test Program</th>
<th>Bug</th>
<th>HaPSet</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>runs</td>
<td>time</td>
<td>runs</td>
<td>time</td>
<td>runs</td>
</tr>
<tr>
<td>MysqlLog</td>
<td>atom</td>
<td>5</td>
<td>0.3</td>
<td>22</td>
<td>1.2</td>
<td>12</td>
</tr>
<tr>
<td>NodeState</td>
<td>order</td>
<td>5</td>
<td>0.3</td>
<td>22</td>
<td>1.5</td>
<td>12</td>
</tr>
<tr>
<td>Loadscript</td>
<td>atom</td>
<td>5</td>
<td>1.0</td>
<td>79</td>
<td>6.0</td>
<td>27</td>
</tr>
<tr>
<td>SeekToItem</td>
<td>atom</td>
<td>3</td>
<td>0.2</td>
<td>127</td>
<td>9.8</td>
<td>22</td>
</tr>
<tr>
<td>UpdateTimer</td>
<td>atom</td>
<td>3</td>
<td>0.2</td>
<td>128</td>
<td>11.2</td>
<td>10</td>
</tr>
<tr>
<td>FileTransport</td>
<td>deadlock</td>
<td>5</td>
<td>0.2</td>
<td>22</td>
<td>1.2</td>
<td>12</td>
</tr>
<tr>
<td>CreateThread</td>
<td>order</td>
<td>5</td>
<td>0.2</td>
<td>22</td>
<td>1.1</td>
<td>12</td>
</tr>
<tr>
<td>ReadWriteProc</td>
<td>order</td>
<td>5</td>
<td>1.9</td>
<td>175</td>
<td>12.0</td>
<td>20</td>
</tr>
<tr>
<td>OpenInputStr</td>
<td>deadlock</td>
<td>5</td>
<td>0.3</td>
<td>1409</td>
<td>107.7</td>
<td>42</td>
</tr>
<tr>
<td>HttpConnect</td>
<td>order</td>
<td>5</td>
<td>3.3</td>
<td>37</td>
<td>28.5</td>
<td>16</td>
</tr>
<tr>
<td>TimerThread</td>
<td>deadlock</td>
<td>3</td>
<td>0.2</td>
<td>101</td>
<td>7.3</td>
<td>18</td>
</tr>
</tbody>
</table>

HaPSet did not miss any bug
Verifying Concurrent Programs

- Concurrency is pervasive, and very difficult to verify
- Active area of research
  - Static analysis, Testing, Dynamic verification, Model checking, …
  - Precise analysis requires reasoning about synchronization
  - Efficient analysis requires controlling complexity of thread interleavings

- Precision AND efficiency of analysis are needed for practical impact
  - Advancements in Decision Procedures (SAT/SMT) offer hope
  - Applications guided by practical concerns (scalability, context bounding)

Great opportunity for industry applications

- Multi-core systems, Many-core systems
- Distributed systems (cloud services, platforms)
(In-house) Source code Verification Service
- Verified source code of several commercial SW projects
- Detected several bugs (not detected by code review or testing) and confirmed

(In-house) Software Factory
- Part of cost reduction plan in a Business Unit
- Verification tools are used for detection of bugs on all projects in a central repository

Acknowledgement: Kishinoue-san, NEC Japan
NEC Labs Verification Group: R&D Overview

Layered Infrastructure across Multiple Projects
- Symbolic constraint solvers
  - perform the search for bugs or proofs
- Verification methods
  - algorithms to formulate the search problem
- Modeling techniques
  - for deriving “effective” verification models from user designs or programs

Application domains
- Software program verification (F-Soft)
- Concurrent system verification (CoBe, Best)
- Embedded/Hybrid Systems (TESSA)

For more information
- http://www.nec-labs.com/~fsoft
- http://www.nec-labs.com

Thank you!

August 2012