Partial Replay of Long-Running Applications

Alvin Cheung
Armando Solar-Lezama
Sam Madden

MIT CSAIL

Bugs are difficult to reproduce

Software Bug Contributed to Blackout

Kevin Poulsen, SecurityFocus 2004-02-11

"It had never evidenced itself until that day," said spokesman Ralph DiNicola. "This fault was so deeply embedded, it took them weeks of poring through millions of lines of code and data to find it."

How Long Did It Take To Fix Bugs?

Sunghun Kim, E. James Whitehead, Jr.

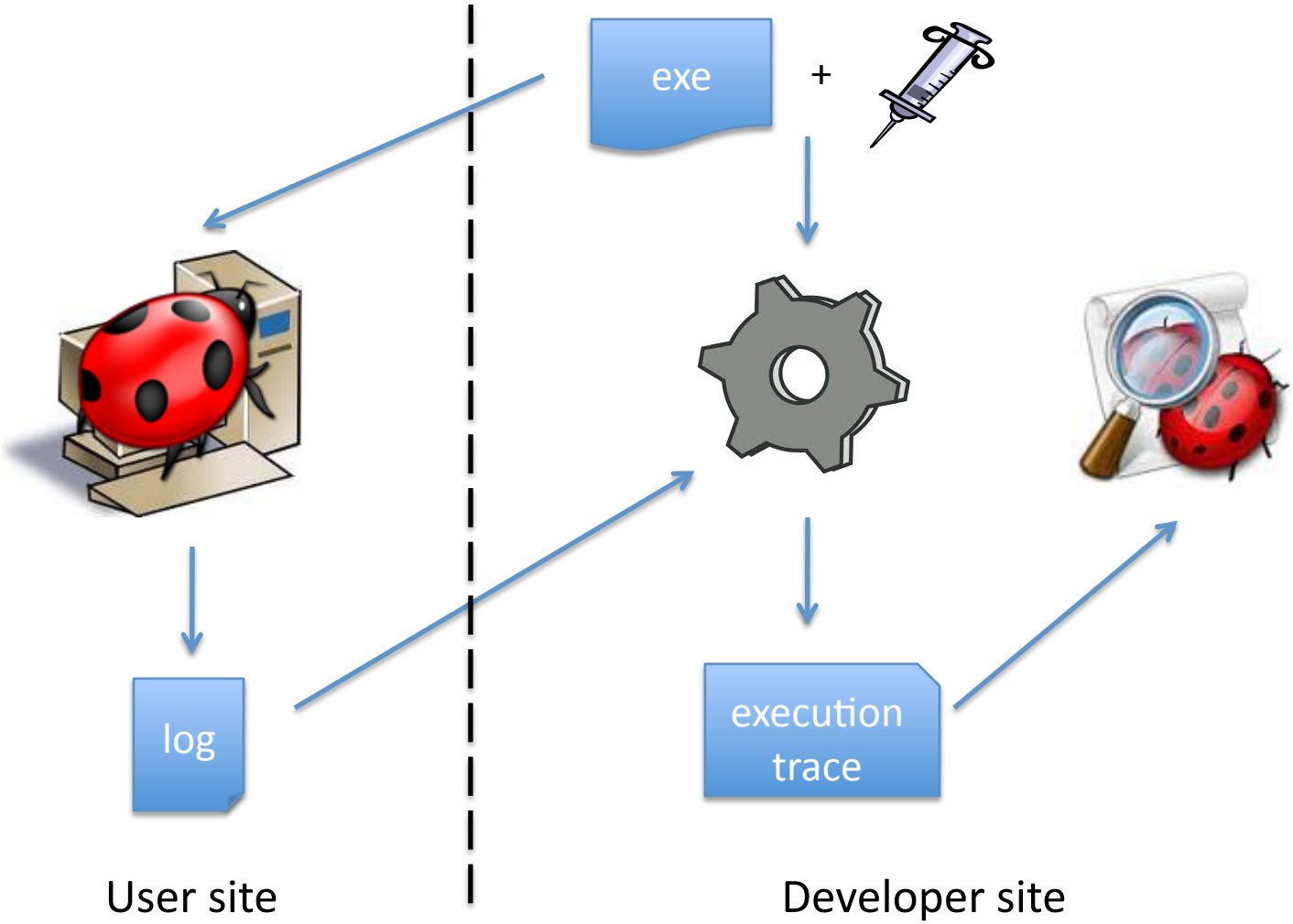
[the plots] show that fixing 50% of the bugs requires appx. 100 to 300 days … The median bug-fix time is about 200 days.

App	# bugs can't be reproduced in bugzilla
gnome	4528
mysql	4175
gentoo	2011
redhat / fedora	2623
firefox	1367
apache	297

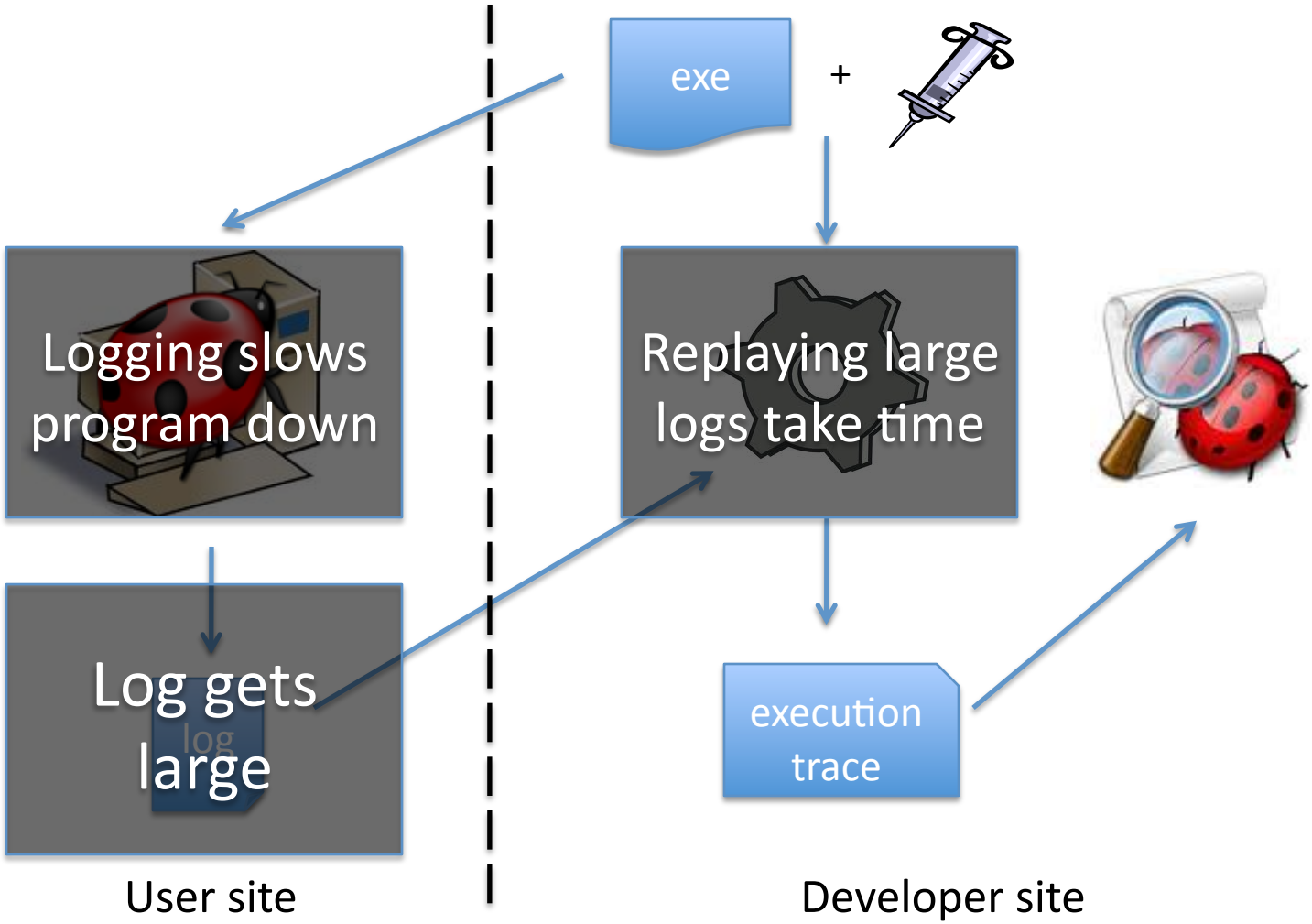
Reproducing Bugs

- Ask user for buggy input
- Guide model checker to find execution trace
  - Non-trivial effort and time
- Use software replayer

Software Replay



Software Replay



bbr to the rescue!

Replayer Wishlist

- Small runtime overhead
- Small log size
- Fast replay time

bbr: A Branch Deterministic Partial Replayer

Small runtime overhead / Small log size

- Record only branches and dynamic array indices
  - Huge log size reduction for data-intensive apps

Fast replay time

- Replay fragment of original execution
- Find execution trace that follows the same control flow path as the original
  - We call that a *branch-deterministic* trace
- Use symbolic execution to find execution trace
  - We call this a *partial symbolic* replay

Running Example

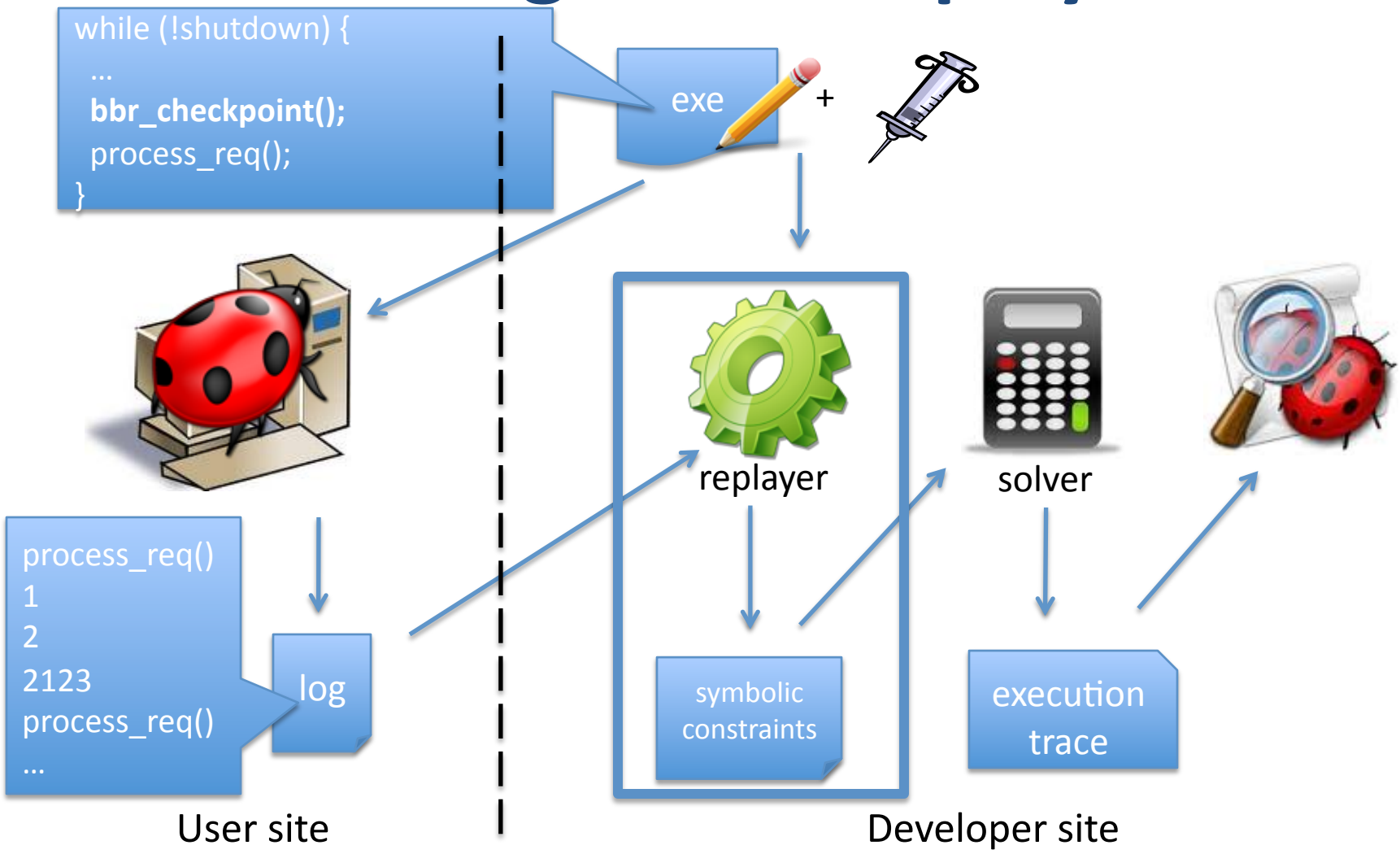
- memcached.c, commit f1f4aec

```
char *do_add_delta (item *it, const int64_t delta)
{
  int64_t value = ITEM_data(it);
  int incr = parse_comma
     ...
   if (incr) {
     ...
   }
   else {
     value -= delta;
     if (value < 0) {
       value = 0;
     }
  ...
```

value and delta originally >= 0

What if value was negative?

Using bbr to replay



Symbolic Execution Example

Code

```
char *do_add_delta (item *it,
                    const int64_t delta)
{
    int64_t value = ITEM_data(it);
    int incr = parse_command(…)

    if (incr) {
        ...
    }
    else {

      value -= delta;


      if (value < 0) {



        value = 0;
      }
```

Symbolic State

`value` → *symVar1*
`incr` → *symVar2*

Log: branch not taken
Add constraint:

symVar2 ≠ 0

`delta` → *symVar3*
`value` → *symVar1 – symVar3*

Log: branch taken

Add constraint:

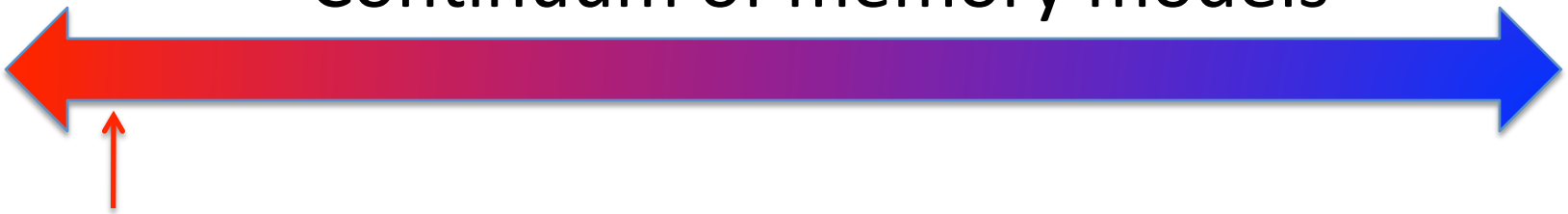
symVar1 – symVar2 < 0

`value` → *0*

bbr internals

Modeling Memory

Continuum of memory models



Entirely symbolic

- Any symbolic variable can be an address
- Rely on solver to find actual values for addresses

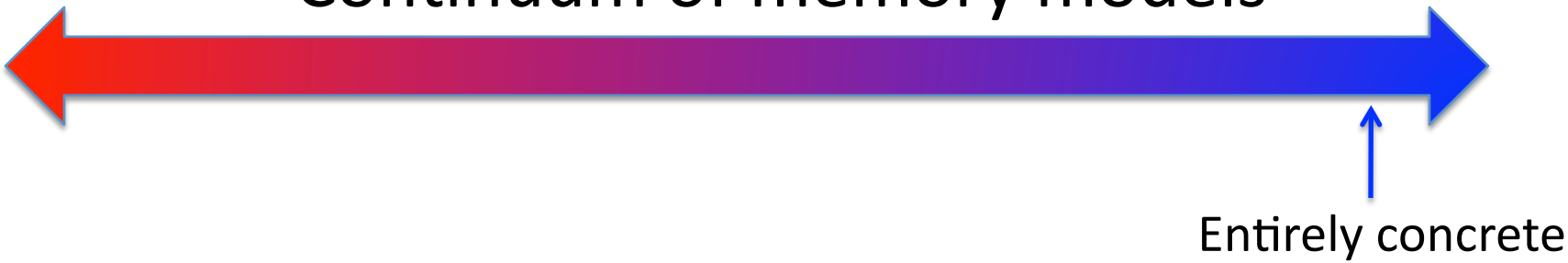
No need to explicitly keep track of aliases

Generate huge constraints with long solve times

Not scalable to replay long executions

Modeling Memory

Continuum of memory models



- All addresses must be concrete values
- Needs complete alias knowledge

Extremely efficient and scalable

Can't do this due to partial replay!

Why do we not have complete alias information?

- Allow replaying of execution fragments
- Access memory locations allocated prior to start of replay
  - We don't know what they point to and their aliasing information

  → Make assumptions about possible aliases

  → Explicitly keep track of may-aliases

  → Ask solver to solve for the actual aliases

Modeling Memory: Our Approach

Continuum of memory models

- More scalable
- Works for partial replay

- Can't replay bugs that rely on unsafe memory operations such as buffer overruns

There are many other tools that target those bugs

Parallel Solver

- Constraints consist of independent groups
  - i.e., do not share any variables
- Split constraints and solve in parallel
  - Substantial savings in solve time

Experiments

Goals

- Runtime overhead
- Log growth rates
- Ability to replay real-world bugs
- Effectiveness of parallel solving

Overhead Experiment

- 6 different long-running apps
- Compared time overhead of native versus 4 different logging mechanisms

  Replay from beginning:

  – *non-det*: log all non-deterministic data

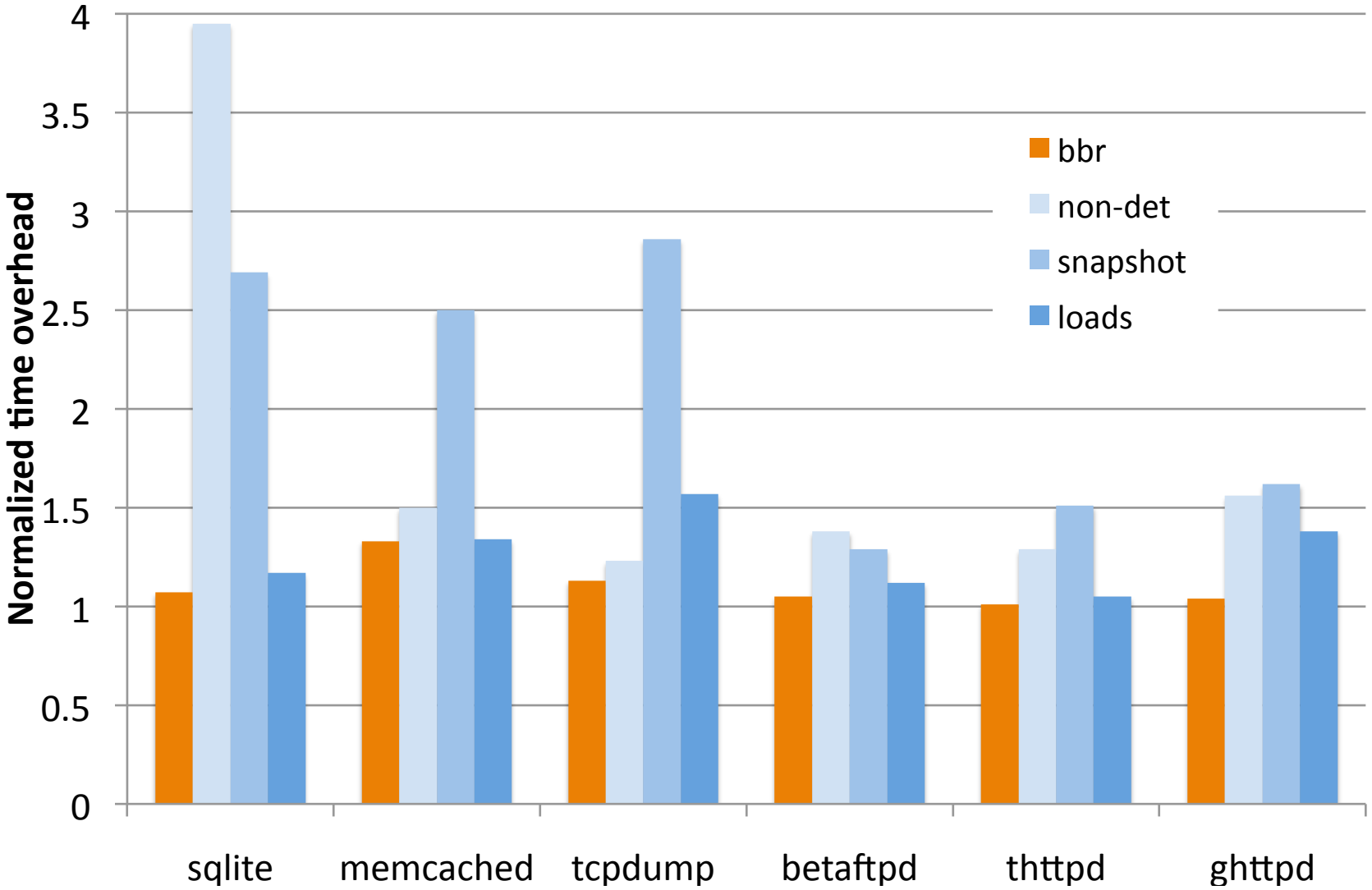
  – *loads*: log values of unpredictable memory loads

  Partial replay:

  – *bbr*: truncates log after N requests

  – *snapshot*: core dumps every N requests + log all non-deterministic data in between

Overhead Experiment



Discussion

- bbr has the lowest time overhead and log growth rate
  - Partially due to data-intensive nature of apps
- Results on CPU-intensive apps were not as good
  - Apps executed many branches

Bug Replay Experiment

- Replayed a total of 11 different real bugs

Bug	LOC	# constraints	Solve time
sqlite cast	2.4M	86k	5hr
memcached CAS	24k	1705	158s
tcpdump ISIS	61M	2.3M	5s
thttpd defang	514k	21k	2s
ghttpd CGI	352k	8k	2s

- Variety of bugs were replayed

Constraint Splitting Experiment

- Replayed different # of requests for two web servers and compared solve times

App (# requests)	# constraints	# groups	Single solve time	Parallel solve time
ghttpd (10)	20383	823	20s	1s
ghttpd (50)	60384	2314	32min	6min
betaftpd (10)	1534	106	2s	1s
Betaftpd (50)	13223	530	40min	13min

- Significant difference in solve times

bbr: a partial replayer using symbolic execution

Low overhead

Small log sizes

Reproduce real-world bugs