CANAL: A Cache Timing Analysis Framework via LLVM Transformation

Chungha Sung | Brandon Paulsen | Chao Wang

ASE 2018
Software verification & analysis

Model Checking

Abstract Interpretation

Symbolic Execution

Checking Functional properties

Ex) assert(x > 1);
Software verification & analysis

Model Checking

Abstract Interpretation

Symbolic Execution

Non-functional Properties e.g. Cache behavior

Ex) The number of cache misses?
You'd have to change each of these tools to model cache behavior.
1. Now, cache (and other non-functional) properties can be handled by existing verifiers
2. General (not tool-specific) cache modeling framework
Overview

Canal process

C/C++ Code

Clang

LLVM Bitcode

LLVM Pass

Code Instrumentation & Cache Computation

Memory Layout

Header generator

cSim.h

Instrumented LLVM Bitcode

Verification tools (SMACK, KLEE, Crab-llvm etc.)

C/C++ Code

Clang

LLVM Bitcode

LLVM Pass

Code Instrumentation & Cache Computation

Memory Layout

Header generator

cSim.h

Instrumented LLVM Bitcode

Verification tools (SMACK, KLEE, Crab-llvm etc.)
Code instrumentation

```cpp
T = Y;
(Inserted function calls below)
__CSIM_Load(address set of "Y", address tags of "Y");
__CSIM_Store(address set of "T", address tags of "T");
```

Code instrumentation is done at the **LLVM-Bitcode** level
Outline

- Motivation

- Code Instrumentation

- **Usages**
  - *Use CANAL as a simulator (omitted)*
  - *Use CANAL with Symbolic execution tool*
  - *Use CANAL with Static analysis tool*
  - *Use CANAL with Software verification tool*

- Conclusion
Usage 1 – Symbolic execution tool

Symbolic execution tools (e.g. Klee)

Check if there exist two inputs that lead to different cache stats (Side-channel leakage)
Usage 1 – Symbolic execution tool (Cont’d)

```c
klee_make_symbolic(&input1);
klee_make_symbolic(&input2);
__CSIM_init_cache();
call_program1(input1);
h1  = __CSIM_num_hit;
m1 = __CSIM_num_miss;
__CSIM_init_cache();
call_program1(input2);
h2  = __CSIM_num_hit;
m2 = __CSIM_num_miss;
assert(h1 == h2 && m1 == m2);
```
Usage 1 – Symbolic execution tool (Cont’d)

```c
klee_make_symbolic(&input1);
klee_make_symbolic(&input2);

__CSIM_init_cache();
call_program1(input1);
h1 = __CSIM_num_hit;
m1 = __CSIM_num_miss;

__CSIM_init_cache();
call_program1(input2);
h2 = __CSIM_num_hit;
m2 = __CSIM_num_miss;

assert(h1 == h2 && m1 == m2);
```

Define symbolic inputs
Usage 1 – Symbolic execution tool (Cont’d)

```c
klee_make_symbolic(&input1);
klee_make_symbolic(&input2);
__CSIM_init_cache();
call_program1(input1);
h1 = __CSIM_num_hit;
m1 = __CSIM_num_miss;
__CSIM_init_cache();
call_program1(input2);
h2 = __CSIM_num_hit;
m2 = __CSIM_num_miss;
assert(h1 == h2 && m1 == m2);
```

Cache status initialization

Run program and get cache stats
Usage 1 – Symbolic execution tool (Cont’d)

klee_make_symbolic(&input1);
klee_make_symbolic(&input2);

__CSIM_init_cache();

call_program1(input1);
h1 = __CSIM_num_hit;
m1 = __CSIM_num_miss;

__CSIM_init_cache();

call_program1(input2);
h2 = __CSIM_num_hit;
m2 = __CSIM_num_miss;

assert(h1 == h2 && m1 == m2);

Cache status initialization

Run program and get cache stats
klee_make_symbolic(&input1);
klee_make_symbolic(&input2);

__CSIM_init_cache();

call_program1(input1);
  h1  = __CSIM_num_hit;
  m1  = __CSIM_num_miss;
__CSIM_init_cache();

call_program1(input2);
  h2  = __CSIM_num_hit;
  m2  = __CSIM_num_miss;

assert(h1 == h2 && m1 == m2);

Check stats are the same
Usage 2 – Software verification tool

Software verification tool (e.g. SMACK)

CANAL

Instrumented LLVM Bitcode

Check if a memory read or write always leads to cach hit/miss (MUST hit/miss analysis)
if (cond)
    buffer[0] = 1;
else
    buffer[16] = 1;

x = buffer[2];
h = __CSIM_Load_ret;
assert (h == true);

Check: Read of buffer[2] always leads to cache hit?
Usage 2 – Software verification tool (Con’d)

```c
if (cond)
    buffer[0] = 1;
else
    buffer[16] = 1;

x = buffer[2];

h = __CSIM_Load_ret;

assert (h == true);
```

buffer[0] and buffer[16] are in different cache line
Usage 2 – Software verification tool (Con’d)

```c
if (cond)
    buffer[0] = 1;
else
    buffer[16] = 1;

x = buffer[2];

h = __CSIM_Load_ret;
assert (h == true);
```

buffer[2] will be the first cache line access when the branch was not taken.
Usage 2 – Software verification tool (Con’d)

if (cond)
    buffer[0] = 1;
else
    buffer[16] = 1;

x = buffer[2];

h = __CSIM_Load_ret;
assert (h == true);

Read the cache status of the last Load/Store operation
Usage 3 – Static analysis tool

**Static analysis tool**
(e.g., Crab-llvm)

**CANAL**

Instrumented LLVM Bitcode

**Static analysis tool**
(e.g., Crab-llvm)

**Compute invariants over cache stats**
(e.g., min/max of cache hits/misses)**
Usage 3 – Static analysis tool (Con’d)

```c
if (cond)
    buffer[0] = 1;
else
    buffer[16] = 1;

buffer[2] = 1;

s_h = __CSIM_num_Store_hit;
s_m = __CSIM_num_Store_miss;

assert (s_h > 1);
assert (s_m < 3);
assert (s_h + s_m == 2);
```
Usage 3 – Static analysis tool (Con’d)

if (cond)
    buffer[0] = 1;
else
    buffer[16] = 1;

buffer[2] = 1;

s_h = __CSIM_num_Store_hit;
s_m = __CSIM_num_Store_miss;

assert (s_h > 1);
assert (s_m < 3);
assert (s_h + s_m == 2);

Check invariants over the number of cache hits and misses.
Conclusions

• Proposed a **unified** framework for modeling cache behaviors through LLVM-transformation

• *CANAL* can be used as a simulator without losing accuracy

• *CANAL* can be used tougher with various software verification tools
Thank you!

https://github.com/canalcache/canal