Transformations and Analysis of LLVM for Model Checking

Vladimír Štill

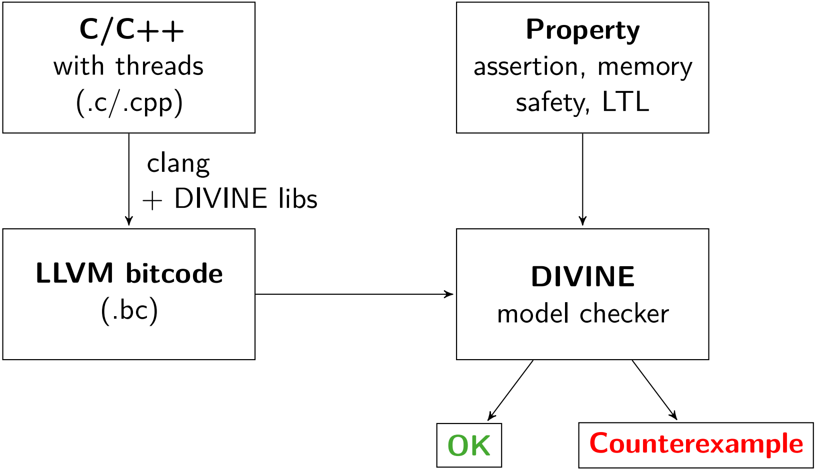
ParaDiSe
Parallel & Distributed Systems Laboratory

Masaryk University

Brno, Czech Republic

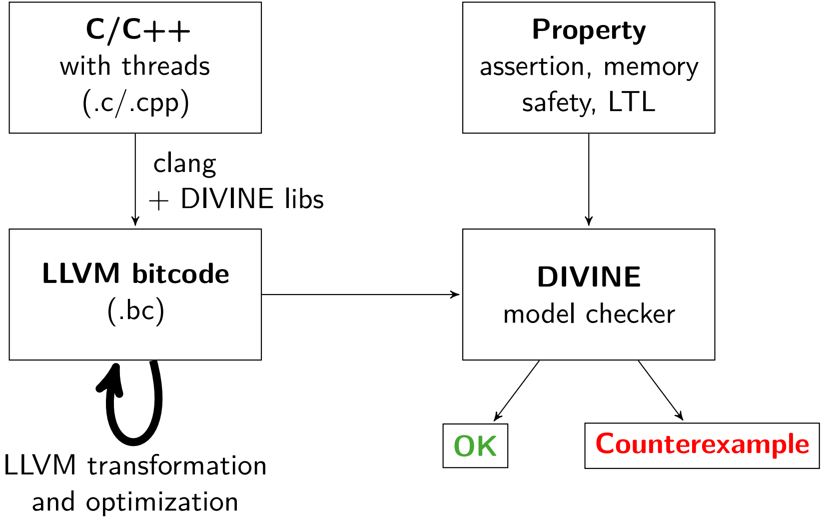
27th November 2015

Introduction



Model checking C++ with DIVINE

Introduction



Adding LLVM-to-LLVM transformations by LART

LART

LLVM Abstraction and Refinement Tool

- cannot do any abstraction or refinement (yet)

LART

LLVM Abstraction and Refinement Tool

- cannot do any abstraction or refinement (yet)

- generic platform for LLVM-to-LLVM transformations
- uses LLVM's C++ API to run transformation and optimization passes

LART

LLVM Abstraction and Refinement Tool

- cannot do any abstraction or refinement (yet)

- generic platform for LLVM-to-LLVM transformations
- uses LLVM's C++ API to run transformation and optimization passes

- currently implements:
  - transformations for SV-COMP
  - weak memory models transformation (from MEMICS 2015)
  - several simple parallel-safe optimizations

SV-COMP

SV-COMP, Concurrency Category

DIVINE is an explicit state model checker

- not really well suited for SV-COMP
- no smart handling of nondeterminism
  - explicit enumeration of nondeterministic choice
- exhaustive search of relevant interleavings
  - but it uses smart reduction techniques to eliminate many uninteresting interleavings

SV-COMP, Concurrency Category

DIVINE is an explicit state model checker

- not really well suited for SV-COMP
- no smart handling of nondeterminism
  - explicit enumeration of nondeterministic choice
- exhaustive search of relevant interleavings
  - but it uses smart reduction techniques to eliminate many uninteresting interleavings

in concurrency category of SV-COMP:

- little or no nondeterminism
- most programs are reasonably small
- DIVINE is quite fast and memory efficient!

How to Verify SV-COMP Models with DIVINE?

bold = LART LLVM transformation

1. compile model with `clang` using `divine compile`
   - this compiles the model and C library functions, `pthreads`,...

How to Verify SV-COMP Models with DIVINE?

bold = LART LLVM transformation

1. compile model with `clang` using `divine compile`
   - this compiles the model and C library functions, `pthreads`,...

2. **add atomic sections to atomic functions**
   - translation from SV-COMP atomic functions to DIVINE's equivalent
   - add atomic masks to atomic functions

How to Verify SV-COMP Models with DIVINE?

bold = LART LLVM transformation

1. compile model with `clang` using `divine compile`
   - this compiles the model and C library functions, `pthreads`,...

2. **add atomic sections to atomic functions**
   - translation from SV-COMP atomic functions to DIVINE's equivalent
   - add atomic masks to atomic functions

3. **change all reads and writes of globals to be volatile**
   - SV-COMP models often have undefined behavior

How to Verify SV-COMP Models with DIVINE?

bold = LART LLVM transformation

1. compile model with `clang` using `divine compile`
   - this compiles the model and C library functions, `pthreads`,...
2. **add atomic sections to atomic functions**
   - translation from SV-COMP atomic functions to DIVINE's equivalent
   - add atomic masks to atomic functions
3. **change all reads and writes of globals to be volatile**
   - SV-COMP models often have undefined behavior
4. run LLVM `opt` to optimize resulting LLVM bitcode (`-Oz`)

How to Verify SV-COMP Models with DIVINE?

5. **try to eliminate or constrain nondeterministic choices**
   - bools are OK
   - nondeterministic pointers can be either `NULL` or `1`
     - we found no models in SV-COMP concurrency category where this causes bad answer
   - for other types try to constrain the choice
     - two common patterns recognised: cast to `bool` and modulo constant value
     - more precise tracking could be done by bounded symbolic execution

How to Verify SV-COMP Models with DIVINE?

5. **try to eliminate or constrain nondeterministic choices**
   - bools are OK
   - nondeterministic pointers can be either `NULL` or `1`
     - we found no models in SV-COMP concurrency category where this causes bad answer
   - for other types try to constrain the choice
     - two common patterns recognised: cast to `bool` and modulo constant value
     - more precise tracking could be done by bounded symbolic execution

6. **disable `malloc` failure**
   - in DIVINE every call to `malloc` can return `NULL` (nondeterminism)
   - SV-COMP, however, seems to work in an idealized world where there is infinite amound of memory. . .

How to Verify SV-COMP Models with DIVINE?

7. run DIVINE, check only for assertions
   - there are often other problems
     - calls with wrong number of parameters
     - missing return statements ($\rightarrow$ undefined value)
   - use state space compression

7. run DIVINE, check only for assertions
   - there are often other problems
     - calls with wrong number of parameters
     - missing return statements ($\rightarrow$ undefined value)
   - use state space compression

Results

We expect to get more then 900 points in concurrency.

- time is the limiting factor

Weak Memory Models

Weak Memory Models

- the order of reads and writes in the code does not need to match the order of their execution
  - compiler optimizations
  - out-of-order execution, cache hierarchy

Weak Memory Models

- the order of reads and writes in the code does not need to match the order of their execution
  - compiler optimizations
  - out-of-order execution, cache hierarchy

- it is hard to reason about memory models
- parallelism is hard even under **Sequential Consistency**
  - reads and writes are immediate and cannot be reordered
  - not realistic, expensive to enforce

Weak Memory Models

- the order of reads and writes in the code does not need to match the order of their execution
  - compiler optimizations
  - out-of-order execution, cache hierarchy

- it is hard to reason about memory models
- parallelism is hard even under **Sequential Consistency**
  - reads and writes are immediate and cannot be reordered
  - not realistic, expensive to enforce

- newer revisions of C/C++ have support for specifying atomic variables with memory access ordering
  - same for Java, LLVM bitcode,...

Weak Memory Models

- the order of reads and writes in the code does not need to match the order of their execution
  - compiler optimizations
  - out-of-order execution, cache hierarchy

- it is hard to reason about memory models
- parallelism is hard even under **Sequential Consistency**
  - reads and writes are immediate and cannot be reordered
  - not realistic, expensive to enforce

- newer revisions of C/C++ have support for specifying atomic variables with memory access ordering
  - same for Java, LLVM bitcode,...

- verifiers often assume sequential consistency
  - so does DIVINE

Total Store Order

- similar to the memory model used by x86_64
- the order of execution of stores is guaranteed to match their order in machine code
  - compiler might still reorder stores
- independent loads can be reordered

Weak Memory Models

```
int x = 0, y = 0;
```

```
void thread0() {
    y = 1;
    cout << "x = " << x;
  }
```

```
void thread1() {
    x = 1;
    cout << "y = " << y;
}
```

Weak Memory Models

```
int x = 0, y = 0;
```

```
void thread0() {
    y = 1;
    cout << "x = " << x;
}
```

```
void thread1() {
    x = 1;
    cout << "y = " << y;
}
```

Total Store Order can be simulated using store buffers:

main memory

0x04	0x08
x = 0	y = 0

thread 0

```
store y 1;
load x;
```

thread 1

```
store x 1;
load y;
```

store buffer for thread 0

store buffer for thread 1


```
int x = 0, y = 0;
```

```
void thread0() {
    y = 1;
    cout << "x = " << x;
}
```

```
void thread1() {
    x = 1;
    cout << "y = " << y;
}
```

Total Store Order can be simulated using store buffers:

main memory

0x04	0x08
x = 0	y = 0

thread 0

```
store y 1;
load x;
```

thread 1

```
store x 1;
load y;
```

store buffer for thread 0

0x08	1	32

store buffer for thread 1

Weak Memory Models

```
int x = 0, y = 0;
```

```
void thread0() {
    y = 1;
    cout << "x = " << x;
}
```

```
void thread1() {
    x = 1;
    cout << "y = " << y;
}
```

Total Store Order can be simulated using store buffers:

main memory

0x04	0x08
x = 0	y = 0

thread 0

```
store y 1;
load x;
```

thread 1

```
store x 1;
load y;
```

store buffer for thread 0

0x08	1	32

store buffer for thread 1

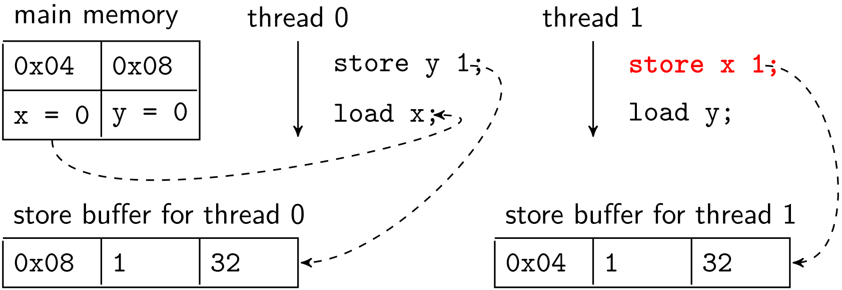
Weak Memory Models

```
int x = 0, y = 0;
```

```
void thread0() {
    y = 1;
    cout << "x = " << x;
  }
```

```
void thread1() {
    x = 1;
    cout << "y = " << y;
}
```

Total Store Order can be simulated using store buffers:



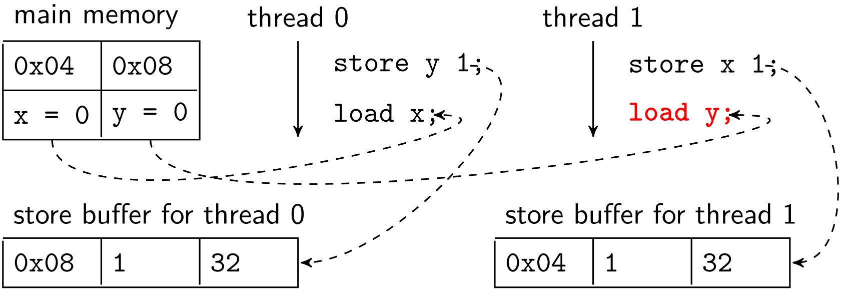
Weak Memory Models

```
int x = 0, y = 0;
```

```
void thread0() {
    y = 1;
    cout << "x = " << x;
  }
```

```
void thread1() {
    x = 1;
    cout << "y = " << y;
}
```

Total Store Order can be simulated using store buffers:



TSO Under-approximation in DIVINE

- we use bounded store buffer to under-approximate TSO
- TSO simulation is implemented as an LLVM-to-LLVM transformation
  - no need to change DIVINE to use weak memory models
  - no need to change verified source code
  - store buffer size can be configured in the transformation

The Transformation to TSO

- every `load`, `store` and memory intrinsic[1] is replaced by function which simulates TSO
- for each thread of the original program, a thread which flushes its store buffer is added
- for an atomic instruction, the store buffer is first flushed and then the instruction is executed without modification

[1] `llvm.memcpy`, `llvm.memmove`, `llvm.memset`

Improvements over the MEMICS Version

- store buffer can be bypassed for load of thread-local memory location
  - the thread-locality is recognized dynamically by DIVINE

Improvements over the MEMICS Version

- store buffer can be bypassed for load of thread-local memory location
  - the thread-locality is recognized dynamically by DIVINE
- manipulations with local variables whose address is never taken are not transformed
  - saves runtime check and entering atomic section (which can cause state to be produced)

Improvements over the MEMICS Version

- store buffer can be bypassed for load of thread-local memory location
  - the thread-locality is recognized dynamically by DIVINE
- manipulations with local variables whose address is never taken are not transformed
  - saves runtime check and entering atomic section (which can cause state to be produced)
- memory safety can be verified
  - entries for memory location which cease to exist are evicted from the store buffer

Memory Safety with TSO

problem: store buffer can be flushed after memory becomes invalid

- flush of value of local variable after function exists
  - remove entry from store buffer before function exit
- flush of value of dynamic memory after `free`
  - remove entry from store buffer in `free`

Memory Safety with TSO

problem: store buffer can be flushed after memory becomes invalid

- flush of value of local variable after function exists
  - **remove entry from store buffer before function exit**
- flush of value of dynamic memory after `free`
  - remove entry from store buffer in `free`

Local Variable Cleanup

remove entry from store buffer before function exit

- easy for C, just add cleanup before `return`

Local Variable Cleanup

remove entry from store buffer before function exit

- easy for C, just add cleanup before `return`
- much harder in C++: **exceptions**
  - function can be exited due to exception propagation
  - cleanups are similar to C++ destructors
  - need to stop the excetpion propagation, do cleanup, resume exception

Improvements over the MEMICS Version

number of states for various models with TSO transformation

Model	MEMICS	+ load private	+ local	SC
`fifo-1`	44 M	5.6 M (7.9×)	1.2 M (4.6×)	7 K
`fifo-2`	338 M	51 M (6.6×)	11 M (4.6×)	7 K
`fifo-3`	672 M	51 M (13×)	11 M (4.6×)	7 K
`simple-1`	538 K	19 K (28×)	11 K (1.7×)	251
`peterson-2`	103 K	40 K (2.6×)	24 K (1.6×)	1.4 K
`pt_mutex-2`	1.6 M	12 K (135×)	7.5 K (1.6×)	98

- load private = loads to memory not visible by other thread bypass store buffer
- local = manipulations with locals to which address is never taken are not transformed

Parallel Safe Optimizations

Parallel Safe Optimizations – Motivation

```
int x = 0;
void *foo( void *_ ) {
    x = 1;
    assert( x == 1 );
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create( &thread, NULL, &foo, NULL );
    x = 2;
    pthread_join( thread, NULL );
}
```

Can the assertion be triggered?

LLVM s `-O0`:

```
define i8* @foo(i8* %_) {
entry:
  %_.addr = alloca i8*
  store i8* %_, i8** %_.addr
  store i32 1, i32* @x
  %0 = load i32, i32* @x
  %tobool = icmp ne i32 %0, 0
  %conv = zext i1 %tobool to i32
  %cmp = icmp eq i32 %conv, 1
  %conv1 = zext i1 %cmp to i32
  call void @__divine_assert(i32 %conv1)
  ret i8* null
}
```

LLVM s -O2:

```
define noalias i8* @foo(i8* %_) {
entry:
  store i32 1, i32* @x
  tail call void @__divine_assert(i32 1)
  ret i8* null
}
```

Parallel Safe Optimizations – Motivation

LLVM s `-O2`:

```
define noalias i8* @foo(i8* %_) {
entry:
  store i32 1, i32* @x
  tail call void @__divine_assert(i32 1)
  ret i8* null
}
```

- optimization changed behavior of the program
- `x` is not set to be `volatile`, or atomic!

Parallel Safe Optimizations

standard compiler optimizations do not preserve parallel behavior of the program

- also do not preserve readability of debugging information
- variables promotion, function inlining, code movement, cycle invariant code motion

Parallel Safe Optimizations

optimizations can increase state space size

- often by adding registers
  - loops

    ```
    while (x != 0) { /* ... */ }
    ```

    can get transformed into

    ```
    if (x != 0) { do { /* ... */ } while (x != 0); }
    ```

  - inlining
  - variable promotion
- states which used to be same are now distinguishable

Parallel Safe Optimizations

- it is desirable to verify code with the optimizations intended by user of verification tool
  - both for usability of results and readability of counterexamples
- but some optimizations can help verifier to run faster, with less memory

Parallel Safe Optimizations

- it is desirable to verify code with the optimizations intended by user of verification tool
  - both for usability of results and readability of counterexamples
- but some optimizations can help verifier to run faster, with less memory

- design optimizations which cannot change verified properties
  - safety, stutter-invariant LTL
- can tightly cooperate with DIVINE

Example of Parallel Safe Optimization

Constant Alloca Elimintation

- local variables in LLVM – results of `alloca` instruction
  - memory for a variable is allocated (on stack) so that it can be passed by referrence
  - often the variable is neither modified nor accessed through pointer
- if variable is constant and does not escape the functions it can be eliminated
  - use of any load from `alloca` is replaced by value which was originally stored into it
- compared to LLVM's register promotion this does not add registers

Atomic Cycles

- if a cycle can be proven to terminate and not perform any visible action it can be executed inside atomic section
- without the atomic section DIVINE emits state after each iteration
- static or dynamic detection of visibility
- need to employ termination detection

Future

Silent loads/stores

- DIVINE's notion of `load`/`store` visibility is based on pointer tracking
- a value might be reachable by pointers from more then one thread, but might be accessed only by one
- mark `load`/`store` as silent to avoid visibility checking
- could also improve verification speed
- requires good pointer analysis

Slicing

- hard to preserve all properties (memory safety,...)
- maybe as an approximation

Future

Symbolic Data

- DIVINE and SymDIVINE will be merged
- symbolic data manipulation compiled in LLVM
- state comparison using SMT in DIVINE
- again, we will need good pointer analysis for parallel code