Dynamic Generation of Likely Invariants for Multithreaded Programs

ICSE 2015

Markus Kusano   Arijit Chattopadhyay   Chao Wang

Virginia Tech, USA

May 21, 2015

Introduction

- "It would be easier to rewrite this entire thing than to understand it"

```
float Q_rsqrt( float number )
{
  long i;
  float x2, y;
  const float threehalfs = 1.5F;

  x2 = number * 0.5F;
  y  = number;
  // evil floating point bit hacking
  i  = * ( long * ) &y;
  // what the %!@*?
  i  = 0x5f3759df - ( i >> 1 );
  y  = * ( float * ) &i;
  // 1st iteration
  y  = y * (threehalfs - (x2*y*y));
  // 2nd iteration,
  //  this can be removed
//  y  = y * ( threehalfs - ( x2 * y *
y ) );

  return y;
}
```

Quake III Arena Source Code

- Often when I start programming on a project with an existing codebase I think "It would be easier to rewrite this entire thing than to understand it."
- For example, consider this fairly infamous piece of code from the Quake 3 arena source
- We've got a function taking in a float; pretty normal
- Next, we've got a floating point value being dereferenced as an integer
- Then, some subtraction and shifting involving a magic number
- Just for fun, we dereference the long back to a float
- And now at this point, I'm completely lost and I think, "Oh great! I'll just read the comments."
- Well. . . we don't even need to go there.
- While this example is a little exaggerated it brings up a good point: knowing the runtime behavior of a program is difficult but essential

Introduction

- "It would be easier to rewrite this entire thing than to understand it"

deref. float as long

```
float Q_rsqrt( float number )
{
  long i;
  float x2, y;
  const float threehalfs = 1.5F;

  x2 = number * 0.5F;
  y  = number;
  // evil floating point bit hacking
  i  = * ( long * ) &y;
  // what the %!@*?
  i  = 0x5f3759df - ( i >> 1 );
  y  = * ( float * ) &i;
  // 1st iteration
  y  = y * (threehalfs - (x2*y*y));
  // 2nd iteration,
  //  this can be removed
//  y  = y * ( threehalfs - ( x2 * y *
y ) );

  return y;
}
```

Quake III Arena Source Code

- Often when I start programming on a project with an existing codebase I think "It would be easier to rewrite this entire thing than to understand it."
- For example, consider this fairly infamous piece of code from the Quake 3 arena source
- We've got a function taking in a float; pretty normal
- Next, we've got a floating point value being dereferenced as an integer
- Then, some subtraction and shifting involving a magic number
- Just for fun, we dereference the long back to a float
- And now at this point, I'm completely lost and I think, "Oh great! I'll just read the comments."
- Well. . . we don't even need to go there.
- While this example is a little exaggerated it brings up a good point: knowing the runtime behavior of a program is difficult but essential

Introduction

- "It would be easier to rewrite this entire thing than to understand it"

deref. float as long

magic number

```
float Q_rsqrt( float number )
{
  long i;
  float x2, y;
  const float threehalfs = 1.5F;

  x2 = number
  y  = number;
  // evil floating point bit hacking
  i  = * ( long * ) &y;
  // what the %!@*?
  i  = 0x5f3759df - ( i >> 1 );
  y  = * ( float * ) &i;
  // 1st iteration
  y  = y * (threehalfs - (x2*y*y));
  // 2nd iteration,
  //  this can be removed
//  y  = y * ( threehalfs - ( x2 * y *
y ) );

  return y;
}
```

Quake III Arena Source Code

- Often when I start programming on a project with an existing codebase I think "It would be easier to rewrite this entire thing than to understand it."
- For example, consider this fairly infamous piece of code from the Quake 3 arena source
- We've got a function taking in a float; pretty normal
- Next, we've got a floating point value being dereferenced as an integer
- Then, some subtraction and shifting involving a magic number
- Just for fun, we dereference the long back to a float
- And now at this point, I'm completely lost and I think, "Oh great! I'll just read the comments."
- Well. . . we don't even need to go there.
- While this example is a little exaggerated it brings up a good point: knowing the runtime behavior of a program is difficult but essential

Introduction

- "It would be easier to rewrite this entire thing than to understand it"

deref. float as long

magic number

deref. long float

```c
float Q_rsqrt( float number )
{
  long i;
  float x2, y;
  const float threehalfs = 1.5F;

  x2 = number
  y  = number;
  // evil floating point bit hacking
  i  = * ( long * ) &y;
  // what the %!@*?
  i  = 0x5f3759df - ( i >> 1 );
  y  = * ( float * ) &i;
  // 1st iteration
  y  = y * (threehalfs - (x2*y*y));
  // 2nd iteration,
  //  this can be removed
//  y  = y * ( threehalfs - ( x2 * y *
y ) );

  return y;
}
```

Quake III Arena Source Code

- Often when I start programming on a project with an existing codebase I think "It would be easier to rewrite this entire thing than to understand it."
- For example, consider this fairly infamous piece of code from the Quake 3 arena source
- We've got a function taking in a float; pretty normal
- Next, we've got a floating point value being dereferenced as an integer
- Then, some subtraction and shifting involving a magic number
- Just for fun, we dereference the long back to a float
- And now at this point, I'm completely lost and I think, "Oh great! I'll just read the comments."
- Well. . . we don't even need to go there.
- While this example is a little exaggerated it brings up a good point: knowing the runtime behavior of a program is difficult but essential

Introduction

- “It would be easier to rewrite this entire thing than to understand it”

```
float Q_rsqrt( float number )
{
  long i;
  float x2, y;
  const float threehalfs = 1.5F;

  x2 = number
  y  = number;
  // evil floating point bit hacking
  i  = * ( long *
  // what the %!@*?
  i  = 0x5f3759df - ( i >> 1 );
  y  = * ( float * ) &i;
  // 1st iteration
  y  = y * (threehalfs - (x2*y*y));
  // 2nd iteration,
  //  this can be removed
//  y  = y * ( threehalfs - ( x2 * y *
y ) );

  return y;
}
```

deref. float as long

magic number

deref. long float

multiplication????

Quake III Arena Source Code

- Often when I start programming on a project with an existing codebase I think “It would be easier to rewrite this entire thing than to understand it.”
- For example, consider this fairly infamous piece of code from the Quake 3 arena source
- We’ve got a function taking in a float; pretty normal
- Next, we’ve got a floating point value being dereferenced as an integer
- Then, some subtraction and shifting involving a magic number
- Just for fun, we dereference the long back to a float
- And now at this point, I’m completely lost and I think, “Oh great! I’ll just read the comments.”
- Well. . . we don’t even need to go there.
- While this example is a little exaggerated it brings up a good point: knowing the runtime behavior of a program is difficult but essential

Introduction

- "It would be easier to rewrite this entire thing than to understand it"
- Knowing the runtime behavior of a program is *essential* but *difficult*

```
float Q_rsqrt( float number )
{
  long i;
  float x2, y;
  const float three        F;

  x2 = number * 0.5F;
  y  = number;
  // evil floating point bit hacking
  i  = * ( long * ) &y;
  // what the %!@*?
  i  = 0x5f3759df - ( i >> 1 );
  y  = * ( float * ) &i;
  // 1st iteration
  y  = y * (threehalfs - (x2*y*y));
  // 2nd iteration,
  //  this can be removed
//  y  = y * ( threehalfs - ( x2 * y *
y ) );

  return y;
}
```

PhD Required

?!?!

A loop?...

Can it really?

Quake III Arena Source Code

- Often when I start programming on a project with an existing codebase I think "It would be easier to rewrite this entire thing than to understand it."
- For example, consider this fairly infamous piece of code from the Quake 3 arena source
- We've got a function taking in a float; pretty normal
- Next, we've got a floating point value being dereferenced as an integer
- Then, some subtraction and shifting involving a magic number
- Just for fun, we dereference the long back to a float
- And now at this point, I'm completely lost and I think, "Oh great! I'll just read the comments."
- Well... we don't even need to go there.
- While this example is a little exaggerated it brings up a good point: knowing the runtime behavior of a program is difficult but essential

Introduction

- Concurrent programs make things even worse
- Are our assumptions correct?
- *Invariants* can provide answers to these questions

```
template <typename T>
class LockFreeQueue {
private:
  struct Node {
    Node( T val ) : value(val), next(nullptr) {
     }
    T value;
    Node* next;
  };
  Node* first;         // for producer only
  atomic<Node*> divider, last;  // shared
void Produce( const T& t ) {
  // add the new item
  last->next = new Node(t);
  last  = last->next; // publish it
  while( first != divider ) { // trim
    Node* tmp = first;
    first = first->next;
    delete tmp;
  }
}
```

Dr. Dobbs, Herb Sutter, Lock-free Queue

The problem is made even more difficult for concurrent programs
Complex interactions between threads are difficult to reason about
Here's is some code, written by a fairly experienced C++ programmer, implementing a lock-free queue
Again, the programmer makes a few assumptions
The pointer to the beginning of the list is not-atomic; are we sure it is only be used by a single producer?
If it is shared, then this non-atomic update is a bug
Pointer's within the list are assumed to be shared? Can we relax this constraint to improve performance
It is difficult, just by examining the source code, to see if these assumptions are correct
For both of these examples, invariants can provide answers to these questions

Introduction

- Concurrent programs make things even worse
- Are our assumptions correct?
- *Invariants* can provide answers to these questions

```
template <typename T>
class LockFreeQueue {
private:
  struct Node {
    Node( T val ) : value(val), next(nullptr) {
    }
    T value;
    Node* next;
  };
  Node* first;         // for producer only
  atomic<Node*> divider, last;  // shared
void Produce( const T& t ) {
  // add the new item
  last->next = new Node(t);
  last  = last->next; // publish it
  while( first != divider ) { // trim
    Node* tmp = first;
    first = first->next;
    delete tmp;
  }
}
```

Is there only one producer??

Dr. Dobbs, Herb Sutter, Lock-free Queue

The problem is made even more difficult for concurrent programs
Complex interactions between threads are difficult to reason about
Here's is some code, written by a fairly experienced C++ programmer, implementing a lock-free queue
Again, the programmer makes a few assumptions
The pointer to the beginning of the list is not-atomic; are we sure it is only be used by a single producer?
If it is shared, then this non-atomic update is a bug
Pointer's within the list are assumed to be shared? Can we relax this constraint to improve performance
It is difficult, just by examining the source code, to see if these assumptions are correct
For both of these examples, invariants can provide answers to these questions

Introduction

- Concurrent programs make things even worse
- Are our assumptions correct?
- *Invariants* can provide answers to these questions

```
template <typename T>
class LockFreeQueue {
private:
  struct Node {
    Node( T val ) : value(val), next(nullptr) {
    }
    T value;
    Node* next;
  };
  Node* first;          // for producer only
  atomic<Node*> divider, last;  // shared
void Produce( const T& t ) {
  // add the new item
  last->next = new Node(t);
  last  = last->next; // publish it
  while( first != divider ) { // trim
    Node* tmp = first;
    first = first->next;
    delete tmp;
  }
}
```

Is there only one producer??

Non-atomic modification

Dr. Dobbs, Herb Sutter, Lock-free Queue

The problem is made even more difficult for concurrent programs
Complex interactions between threads are difficult to reason about
Here's is some code, written by a fairly experienced C++ programmer, implementing a lock-free queue
Again, the programmer makes a few assumptions
The pointer to the beginning of the list is not-atomic; are we sure it is only be used by a single producer?
If it is shared, then this non-atomic update is a bug
Pointer's within the list are assumed to be shared? Can we relax this constraint to improve performance
It is difficult, just by examining the source code, to see if these assumptions are correct
For both of these examples, invariants can provide answers to these questions

Introduction

- Concurrent programs make things even worse
- Are our assumptions correct?
- *Invariants* can provide answers to these questions

```
template <typename T>
class LockFreeQueue {
private:
  struct Node {
    Node( T val ) : value(val), next(nullptr) {
    }
    T value;
    Node* next;
  };
  Node* first;          // for producer only
  atomic<Node*> divider, last;  // shared
void Produce( const T& t ) {
  // add the new item
  last->next = new Node(t);
  last   = last->next; // publish it
  while( first != divider ) { // trim
    Node* tmp = first;
    first = first->next;
    delete tmp;
  }
}
```

Is this really shared?

Is there only one producer??

Non-atomic modification

Dr. Dobbs, Herb Sutter, Lock-free Queue

The problem is made even more difficult for concurrent programs
Complex interactions between threads are difficult to reason about
Here's is some code, written by a fairly experienced C++ programmer, implementing a lock-free queue
Again, the programmer makes a few assumptions
The pointer to the beginning of the list is not-atomic; are we sure it is only be used by a single producer?
If it is shared, then this non-atomic update is a bug
Pointer's within the list are assumed to be shared? Can we relax this constraint to improve performance
It is difficult, just by examining the source code, to see if these assumptions are correct
For both of these examples, invariants can provide answers to these questions

Contribution: Udon

- Udon: Multithreaded Dynamic Invariant Generator
- Statistical inference + stateless model checking
- Dynamically explores thread schedules
- Potential Uses:
  - Program understanding
  - Fault localization
  - Automatic repair
- LLVM based: faster

This brings us to our contribution
We present Udon, a Dynamic Likely Invariant generator for Multithreaded Programs
Udon is a novel combination of statistical inference and stateless model checking
Udon automatically searches through different schedules to generate invariants
Sequential invariant generation tools have been widely successful.
Udon fills the gap by allowing them to handle multithreaded programs
For example, we already showed how dynamic invariants could help in program understanding
It has also been used in the past in sequential programs for fault localization and automatic repair
Additionally, we used LLVM for instrumentation which resulted in our method being faster and more accurate than prior work

Overview

Next, I will go over both true and likely invariants

Invariants

- An invariant is a truth condition at a particular location

```
int inc(int i) {
  ret = i + 1;
  return ret;
}


void inc_ptr(int *p) {
  ret = p) + 1;
```


We make a distinction between invariants and likely invariants.
An invariant is a proposition which is true on all possible paths through the program
In other words, it holds for all possible inputs of the program
Consider this simple function which increments its input
One useful invariant about this function is that its output is always greater than its input
This is a nice compact summary of the functions behavior
Next, consider this function which indirectly increments a value through a pointer
An invariant stating that p can never be NULL provides a safety property: on all paths through the program, this function will never have a NULL pointer violation

Invariants

- An invariant is a truth condition at a particular location
- It is true on *all* possible paths of the program

```
int inc(int i) {
  ret = i + 1;
  return ret;
}


void inc_ptr(int *p) {
  ret = p) + 1;
```


We make a distinction between invariants and likely invariants.
An invariant is a proposition which is true on all possible paths through the program
In other words, it holds for all possible inputs of the program
Consider this simple function which increments its input
One useful invariant about this function is that its output is always greater than its input
This is a nice compact summary of the functions behavior
Next, consider this function which indirectly increments a value through a pointer
An invariant stating that p can never be NULL provides a safety property: on all paths through the program, this function will never have a NULL pointer violation

Invariants

- An invariant is a truth condition at a particular location
- It is true on *all* possible paths of the program

```
int inc(int i) {
  ret = i + 1;
  return ret;
}
```

ret > x

```
void inc_ptr(int *p) {
  ret = p) + 1;
```


We make a distinction between invariants and likely invariants.
An invariant is a proposition which is true on all possible paths through the program
In other words, it holds for all possible inputs of the program
Consider this simple function which increments its input
One useful invariant about this function is that its output is always greater than its input
This is a nice compact summary of the functions behavior
Next, consider this function which indirectly increments a value through a pointer
An invariant stating that p can never be NULL provides a safety property: on all paths through the program, this function will never have a NULL pointer violation

Invariants

- An invariant is a truth condition at a particular location
- It is true on *all* possible paths of the program

```
int inc(int i) {
  ret = i + 1;
  return ret;
}
```

$ret > x$

```
void inc_ptr(int *p) {
  ret = p) + 1;
```


We make a distinction between invariants and likely invariants.
An invariant is a proposition which is true on all possible paths through the program
In other words, it holds for all possible inputs of the program
Consider this simple function which increments its input
One useful invariant about this function is that its output is always greater than its input
This is a nice compact summary of the functions behavior
Next, consider this function which indirectly increments a value through a pointer
An invariant stating that p can never be NULL provides a safety property: on all paths through the program, this function will never have a NULL pointer violation

Invariants

- An invariant is a truth condition at a particular location
- It is true on *all* possible paths of the program

```
int inc(int i) {
  ret = i + 1;
  return ret;
}
```

$ret > x$

```
void inc_ptr(int *p
  ret = p) + 1;
```

$p \neq NULL$

We make a distinction between invariants and likely invariants.
An invariant is a proposition which is true on all possible paths through the program
In other words, it holds for all possible inputs of the program
Consider this simple function which increments its input
One useful invariant about this function is that its output is always greater than its input
This is a nice compact summary of the functions behavior
Next, consider this function which indirectly increments a value through a pointer
An invariant stating that p can never be NULL provides a safety property: on all paths through the program, this function will never have a NULL pointer violation

Likely Invariants

- A likely invariant is a truth condition at a particular location

```
int inc(int i) {
  ret = i + 1;
  return ret;
}


void inc_ptr(int *p) {
  ret = p) + 1;
```


On the other hand, a likely invariant is a proposition which is true given a set of test inputs to the program
It may not be true for all possible inputs
If we examine the same two functions, consider that given two test inputs to the program results in the following values being passed to the functions
Given these values, we can generate some likely invariants.
For instance, the increment operation returns either 4 or 8
And, the input to the pointer increment operation is still never NULL
The usefulness in dynamic likely invariant generation is scalability: not all program paths need to be explored
However, The invariants almost match the true program invariants so there is a slight tradeoff in accuracy for scalability

Likely Invariants

- A likely invariant is a truth condition at a particular location
- It is true on *some* possible paths of the program

```
int inc(int i) {
  ret = i + 1;
  return ret;
}


void inc_ptr(int *p) {
  ret = p) + 1;
```


On the other hand, a likely invariant is a proposition which is true given a set of test inputs to the program
It may not be true for all possible inputs
If we examine the same two functions, consider that given two test inputs to the program results in the following values being passed to the functions
Given these values, we can generate some likely invariants.
For instance, the increment operation returns either 4 or 8
And, the input to the pointer increment operation is still never NULL
The usefulness in dynamic likely invariant generation is scalability: not all program paths need to be explored
However, The invariants almost match the true program invariants so there is a slight tradeoff in accuracy for scalability

Likely Invariants

- A likely invariant is a truth condition at a particular location
- It is true on *some* possible paths of the program
  - inc(3), inc_ptr(0x0AB7FC5B)

```
int inc(int i) {
  ret = i + 1;
  return ret;
}


void inc_ptr(int *p) {
  ret = p) + 1;
```


On the other hand, a likely invariant is a proposition which is true given a set of test inputs to the program
It may not be true for all possible inputs
If we examine the same two functions, consider that given two test inputs to the program results in the following values being passed to the functions
Given these values, we can generate some likely invariants.
For instance, the increment operation returns either 4 or 8
And, the input to the pointer increment operation is still never NULL
The usefulness in dynamic likely invariant generation is scalability: not all program paths need to be explored
However, The invariants almost match the true program invariants so there is a slight tradeoff in accuracy for scalability

Likely Invariants

- A likely invariant is a truth condition at a particular location
- It is true on *some* possible paths of the program
  - inc(3), inc_ptr(0x0AB7FC5B)
  - inc(7), inc_ptr(0x0AB7F194)

```
int inc(int i) {
  ret = i + 1;
  return ret;
}


void inc_ptr(int *p) {
  ret = p) + 1;
```


On the other hand, a likely invariant is a proposition which is true given a set of test inputs to the program
It may not be true for all possible inputs
If we examine the same two functions, consider that given two test inputs to the program results in the following values being passed to the functions
Given these values, we can generate some likely invariants.
For instance, the increment operation returns either 4 or 8
And, the input to the pointer increment operation is still never NULL
The usefulness in dynamic likely invariant generation is scalability: not all program paths need to be explored
However, The invariants almost match the true program invariants so there is a slight tradeoff in accuracy for scalability

Likely Invariants

- A likely invariant is a truth condition at a particular location
- It is true on *some* possible paths of the program
  - `inc(3)`, `inc_ptr(0x0AB7FC5B)`
  - `inc(7)`, `inc_ptr(0x0AB7F194)`

```
int inc(int i) {
  ret = i + 1;
  return ret;
}
```

$ret = \{ 4, 8 \}$

```
void inc_ptr(int *p) {
  ret = p) + 1;
```


On the other hand, a likely invariant is a proposition which is true given a set of test inputs to the program
It may not be true for all possible inputs
If we examine the same two functions, consider that given two test inputs to the program results in the following values being passed to the functions
Given these values, we can generate some likely invariants.
For instance, the increment operation returns either 4 or 8
And, the input to the pointer increment operation is still never NULL
The usefulness in dynamic likely invariant generation is scalability: not all program paths need to be explored
However, The invariants almost match the true program invariants so there is a slight tradeoff in accuracy for scalability

Likely Invariants

- A likely invariant is a truth condition at a particular location
- It is true on *some* possible paths of the program
  - `inc(3)`, `inc_ptr(0x0AB7FC5B)`
  - `inc(7)`, `inc_ptr(0x0AB7F194)`

```
int inc(int i) {
  ret = i + 1;
  return ret;
}
```

$ret = \{ 4, 8 \}$

```
void inc_ptr(int *p
  ret = p) + 1;
```

$p \neq NULL$

On the other hand, a likely invariant is a proposition which is true given a set of test inputs to the program
It may not be true for all possible inputs
If we examine the same two functions, consider that given two test inputs to the program results in the following values being passed to the functions
Given these values, we can generate some likely invariants.
For instance, the increment operation returns either 4 or 8
And, the input to the pointer increment operation is still never NULL
The usefulness in dynamic likely invariant generation is scalability: not all program paths need to be explored
However, The invariants almost match the true program invariants so there is a slight tradeoff in accuracy for scalability

Likely Invariants

- A likely invariant is a truth condition at a particular location
- It is true on *some* possible paths of the program
  - `inc(3)`, `inc_ptr(0x0AB7FC5B)`
  - `inc(7)`, `inc_ptr(0x0AB7F194)`
- Why? Generating likely invariants is scalable [Ernst et al., 2007]

```
int inc(int i) {
  ret = i + 1;
  return ret;
}
```

$ret = \{4, 8\}$

```
void inc_ptr(int *p
  ret = p) + 1;
```

$p \neq NULL$

On the other hand, a likely invariant is a proposition which is true given a set of test inputs to the program
It may not be true for all possible inputs
If we examine the same two functions, consider that given two test inputs to the program results in the following values being passed to the functions
Given these values, we can generate some likely invariants.
For instance, the increment operation returns either 4 or 8
And, the input to the pointer increment operation is still never NULL
The usefulness in dynamic likely invariant generation is scalability: not all program paths need to be explored
However, The invariants almost match the true program invariants so there is a slight tradeoff in accuracy for scalability

Overview

Introduction

Background

Invariants

Systematic Stateless Model Checking

Udon

Experimental Results

Next, I will provide a brief introduction to stateless model checking of concurrent programs

Stateless Model Checking of Concurrent Programs

- Explore a subset of all possible executions

```
int x,y;
void thread1(void) {
  x = 1;
  y = 1;
}
void thread2(void) {
  y = 2;
}
```


Udon uses a stateless model checker to intelligently explore different thread schedules
The goal of a stateless model checker is to explore a subset of all the possible combinations of thread schedules
For example, in this program, there are three different thread schedules.
One where thread one runs first followed by thread two, one where they both interleave, and one where thread 2 runs first followed by thread 1
The search strategy used in the stateless model checker determines which subset of these exeuctions are chosen

Stateless Model Checking of Concurrent Programs

- Explore a subset of all possible executions
- Possible Executions:
  - x = 1; y = 1; y = 2

```
int x,y;
void thread1(void) {
  x = 1;
  y = 1;
}
void thread2(void) {
  y = 2;
}
```


Udon uses a stateless model checker to intelligently explore different thread schedules
The goal of a stateless model checker is to explore a subset of all the possible combinations of thread schedules
For example, in this program, there are three different thread schedules.
One where thread one runs first followed by thread two, one where they both interleave, and one where thread 2 runs first followed by thread 1
The search strategy used in the stateless model checker determines which subset of these exeuctions are chosen

Stateless Model Checking of Concurrent Programs

- Explore a subset of all possible executions
- Possible Executions:
  - x = 1; y = 1; y = 2
  - x = 1; y = 2; y = 1

```
int x,y;
void thread1(void) {
  x = 1;
  y = 1;
}
void thread2(void) {
  y = 2;
}
```


Udon uses a stateless model checker to intelligently explore different thread schedules
The goal of a stateless model checker is to explore a subset of all the possible combinations of thread schedules
For example, in this program, there are three different thread schedules.
One where thread one runs first followed by thread two, one where they both interleave, and one where thread 2 runs first followed by thread 1
The search strategy used in the stateless model checker determines which subset of these exeuctions are chosen

Stateless Model Checking of Concurrent Programs

- Explore a subset of all possible executions
- Possible Executions:
  - `x = 1; y = 1; y = 2`
  - `x = 1; y = 2; y = 1`
  - `y = 2; x = 1; y = 1`

```
int x,y;
void thread1(void) {
  x = 1;
  y = 1;
}
void thread2(void) {
  y = 2;
}
```


Udon uses a stateless model checker to intelligently explore different thread schedules
The goal of a stateless model checker is to explore a subset of all the possible combinations of thread schedules
For example, in this program, there are three different thread schedules.
One where thread one runs first followed by thread two, one where they both interleave, and one where thread 2 runs first followed by thread 1
The search strategy used in the stateless model checker determines which subset of these exeuctions are chosen

Stateless Model Checking of Concurrent Programs

- Explore a subset of all possible executions
- Possible Executions:
  - `x = 1; y = 1; y = 2`
  - `x = 1; y = 2; y = 1`
  - `y = 2; x = 1; y = 1`
- Explore a minimal subset of all thread schedules [Godefroid, 1997]

```
int x,y;
void thread1(void) {
  x = 1;
  y = 1;
}
void thread2(void) {
  y = 2;
}
```


Udon uses a stateless model checker to intelligently explore different thread schedules
The goal of a stateless model checker is to explore a subset of all the possible combinations of thread schedules
For example, in this program, there are three different thread schedules.
One where thread one runs first followed by thread two, one where they both interleave, and one where thread 2 runs first followed by thread 1
The search strategy used in the stateless model checker determines which subset of these exeuctions are chosen

Thread Schedule Search Strategies

- Dynamic Partial Order Reduction (DPOR): Guaranteed to explore all thread schedules relevant to safety properties and deadlocks [Flanagan and Godefroid, 2005]

Simpsons.wikia.com

Udon can use three different search strategies
The first is dynamic partial order reduction. It is theoretically sound in that it is guaranteed to explore all the concurrent behaviors of a program
Even though it is sound, it is capable of offering a significant reduction by using some complex math we will not get into here
However, two heuristic based approaches aim to explore even less of the concurrent state space but still find bugs
The first is preemptive context bounding. This heuristic bounds the number of times a schedule can context switch between threads
The second is happy set: it explores a subset of memory access orderings
Both preemptive context bounding and happy set have been shown, empirically, to detect bugs faster than DPOR even though they are unsound.

Thread Schedule Search Strategies

- Dynamic Partial Order Reduction (DPOR): Guaranteed to explore all thread schedules relevant to safety properties and deadlocks [Flanagan and Godefroid, 2005]
- Preemptive Context Bounding (PCB): limit number of scheduler context switches [Musuvathi and Qadeer, 2007]

Simpsons.wikia.com

Udon can use three different search strategies
The first is dynamic partial order reduction. It is theoretically sound in that it is guaranteed to explore all the concurrent behaviors of a program
Even though it is sound, it is capable of offering a significant reduction by using some complex math we will not get into here
However, two heuristic based approaches aim to explore even less of the concurrent state space but still find bugs
The first is preemptive context bounding. This heuristic bounds the number of times a schedule can context switch between threads
The second is happy set: it explores a subset of memory access orderings
Both preemptive context bounding and happy set have been shown, empirically, to detect bugs faster than DPOR even though they are unsound.

Thread Schedule Search Strategies

- Dynamic Partial Order Reduction (DPOR): Guaranteed to explore all thread schedules relevant to safety properties and deadlocks [Flanagan and Godefroid, 2005]
- Preemptive Context Bounding (PCB): limit number of scheduler context switches [Musuvathi and Qadeer, 2007]
- History Aware Predecessor Set (HaPSet): explore subset of memory access orderings [Wang et al., 2011]

Simpsons.wikia.com

Udon can use three different search strategies
The first is dynamic partial order reduction. It is theoretically sound in that it is guaranteed to explore all the concurrent behaviors of a program
Even though it is sound, it is capable of offering a significant reduction by using some complex math we will not get into here
However, two heuristic based approaches aim to explore even less of the concurrent state space but still find bugs
The first is preemptive context bounding. This heuristic bounds the number of times a schedule can context switch between threads
The second is happy set: it explores a subset of memory access orderings
Both preemptive context bounding and happy set have been shown, empirically, to detect bugs faster than DPOR even though they are unsound.

Overview

Next, I will discuss our new tool, Udon

Likely Invariant Generation of Concurrent Programs

- Why Not just use Daikon?
  [Ernst et al., 2007]

```
int balance = 400;
int getBalance() {
  int bal;
  Lock();
  bal = balance;
  Unlock();
  return bal;
}
void setBalance(int bal) {
  Lock();
  balance = bal;
  Unlock();
}
void withdraw() {
  int bal = getBalance();
  newBal = bal - 100;
  setBalance(newBal);
}
int main(void) {
  thread_create(&t1,withdraw);
  thread_create(&t2,withdraw);
  thread_join(t1);
  thread_join(t1);
  assert(balance==200);
}
```


This example shows why using Daikon on concurrent programs is not acurrate
The main reason is that a naive exploration of the state space often misses concurrent behavior
In this program, two threads are concurrently modifying a shared variable balance initialized to 400.
The getBalance function is an atomic read, and the setBalance function is an atomic write
However, the withdraw function is a non-atomic read-modify-write
As a result, when two threads are concurrently withdrawing 100, the final state can be either 300 or 200
Since the bug is often missed, the incorrect invariant that $newBal = bal - 100$ in the withdraw function is often reported by Daikon.
The reason for this is that the kernel timeslice is often long enough such that the two threads do not interfere with each other
Reporting this to the developer implies that the update in withdraw is atomic
However, Udon, which uses a systematic exploration of the state space produces the correct invariant: newBal ¡ bal
We refer to these invariants over blocks of code as transition invariants. We believe they are particularly useful in analyzing multithreaded programs.

Likely Invariant Generation of Concurrent Programs

- Why Not just use Daikon? [Ernst et al., 2007]
- Naive exploration of the state space often misses the bug

```
int balance = 400;
int getBalance() {
  int bal;
  Lock();
  bal = balance;
  Unlock();
  return bal;
}
void setBalance(int bal) {
  Lock();
  balance = bal;
  Unlock();
}
void withdraw() {
  int bal = getBalance();
  newBal = bal - 100;
  setBalance(newBal);
}
int main(void) {
  thread_create(&t1,withdraw);
  thread_create(&t2,withdraw);
  thread_join(t1);
  thread_join(t1);
  assert(balance==200);
}
```


This example shows why using Daikon on concurrent programs is not acurrate
The main reason is that a naive exploration of the state space often misses concurrent behavior
In this program, two threads are concurrently modifying a shared variable balance initialized to 400.
The getBalance function is an atomic read, and the setBalance function is an atomic write
However, the withdraw function is a non-atomic read-modify-write
As a result, when two threads are concurrently withdrawing 100, the final state can be either 300 or 200
Since the bug is often missed, the incorrect invariant that $newBal = bal - 100$ in the withdraw function is often reported by Daikon.
The reason for this is that the kernel timeslice is often long enough such that the two threads do not interfere with each other
Reporting this to the developer implies that the update in withdraw is atomic
However, Udon, which uses a systematic exploration of the state space produces the correct invariant: newBal ¡ bal
We refer to these invariants over blocks of code as transition invariants. We believe they are particularly useful in analyzing multithreaded programs.

Likely Invariant Generation of Concurrent Programs

- Why Not just use Daikon? [Ernst et al., 2007]
- Naive exploration of the state space often misses the bug

Shared variable

```
int balance = 400;
int getBalance() {
  int bal;
  Lock();
  bal = balance;
  Unlock();
  return bal;
}
void setBalance(int bal) {
  Lock();
  balance = bal;
  Unlock();
}
void withdraw() {
  int bal = getBalance();
  newBal = bal - 100;
  setBalance(newBal);
}
int main(void) {
  thread_create(&t1,withdraw);
  thread_create(&t2,withdraw);
  thread_join(t1);
  thread_join(t1);
  assert(balance==200);
}
```


This example shows why using Daikon on concurrent programs is not acurrate
The main reason is that a naive exploration of the state space often misses concurrent behavior
In this program, two threads are concurrently modifying a shared variable balance initialized to 400.
The getBalance function is an atomic read, and the setBalance function is an atomic write
However, the withdraw function is a non-atomic read-modify-write
As a result, when two threads are concurrently withdrawing 100, the final state can be either 300 or 200
Since the bug is often missed, the incorrect invariant that $newBal = bal - 100$ in the withdraw function is often reported by Daikon.
The reason for this is that the kernel timeslice is often long enough such that the two threads do not interfere with each other
Reporting this to the developer implies that the update in withdraw is atomic
However, Udon, which uses a systematic exploration of the state space produces the correct invariant: newBal ¡ bal
We refer to these invariants over blocks of code as transition invariants. We believe they are particularly useful in analyzing multithreaded programs.

Likely Invariant Generation of Concurrent Programs

- Why Not just use Daikon? [Ernst et al., 2007]
- Naive exploration of the state space often misses the bug

Shared variable

Atomic read

```
int balance = 400;
int getBalance() {
  int bal;
  Lock();
  bal = balance;
  Unlock();
  return bal;
}
void setBalance(int bal) {
  Lock();
  balance = bal;
  Unlock();
}
void withdraw() {
  int bal = getBalance();
  newBal = bal - 100;
  setBalance(newBal);
}
int main(void) {
  thread_create(&t1,withdraw);
  thread_create(&t2,withdraw);
  thread_join(t1);
  thread_join(t1);
  assert(balance==200);
}
```


This example shows why using Daikon on concurrent programs is not acurrate
The main reason is that a naive exploration of the state space often misses concurrent behavior
In this program, two threads are concurrently modifying a shared variable balance initialized to 400.
The getBalance function is an atomic read, and the setBalance function is an atomic write
However, the withdraw function is a non-atomic read-modify-write
As a result, when two threads are concurrently withdrawing 100, the final state can be either 300 or 200
Since the bug is often missed, the incorrect invariant that $newBal = bal - 100$ in the withdraw function is often reported by Daikon.
The reason for this is that the kernel timeslice is often long enough such that the two threads do not interfere with each other
Reporting this to the developer implies that the update in withdraw is atomic
However, Udon, which uses a systematic exploration of the state space produces the correct invariant: newBal ¡ bal
We refer to these invariants over blocks of code as transition invariants. We believe they are particularly useful in analyzing multithreaded programs.

Likely Invariant Generation of Concurrent Programs

- Why Not just use Daikon? [Ernst et al., 2007]
- Naive exploration of the state space often misses the bug

Shared variable

Atomic read

Atomic write

```
int balance = 400;
int getBalance() {
  int bal;
  Lock();
  bal = balance;
  Unlock();
  return bal;
}
void setBalance(int bal) {
  Lock();
  balance = bal;
  Unlock();
}
void withdraw() {
  int bal = getBalance();
  newBal = bal - 100;
  setBalance(newBal);
}
int main(void) {
  thread_create(&t1,withdraw);
  thread_create(&t2,withdraw);
  thread_join(t1);
  thread_join(t1);
  assert(balance==200);
}
```


This example shows why using Daikon on concurrent programs is not acurrate
The main reason is that a naive exploration of the state space often misses concurrent behavior
In this program, two threads are concurrently modifying a shared variable balance initialized to 400.
The getBalance function is an atomic read, and the setBalance function is an atomic write
However, the withdraw function is a non-atomic read-modify-write
As a result, when two threads are concurrently withdrawing 100, the final state can be either 300 or 200
Since the bug is often missed, the incorrect invariant that $newBal = bal - 100$ in the withdraw function is often reported by Daikon.
The reason for this is that the kernel timeslice is often long enough such that the two threads do not interfere with each other
Reporting this to the developer implies that the update in withdraw is atomic
However, Udon, which uses a systematic exploration of the state space produces the correct invariant: newBal ¡ bal
We refer to these invariants over blocks of code as transition invariants. We believe they are particularly useful in analyzing multithreaded programs.

Likely Invariant Generation of Concurrent Programs

- Why Not just use Daikon? [Ernst et al., 2007]
- Naive exploration of the state space often misses the bug

Shared variable

Atomic read

Atomic write

Non-*atomic* read-modify-write

```
int balance = 400;
int getBalance() {
  int bal;
  Lock();
  bal = balance;
  Unlock();
  return bal;
}
void setBalance(int bal) {
  Lock();
  balance = bal;
  Unlock();
}
void withdraw() {
  int bal = getBalance();
  newBal = bal - 100;
  setBalance(newBal);
}
int main(void) {
  thread_create(&t1,withdraw);
  thread_create(&t2,withdraw);
  thread_join(t1);
  thread_join(t1);
  assert(balance==200);
}
```


This example shows why using Daikon on concurrent programs is not acurrate
The main reason is that a naive exploration of the state space often misses concurrent behavior
In this program, two threads are concurrently modifying a shared variable balance initialized to 400.
The getBalance function is an atomic read, and the setBalance function is an atomic write
However, the withdraw function is a non-atomic read-modify-write
As a result, when two threads are concurrently withdrawing 100, the final state can be either 300 or 200
Since the bug is often missed, the incorrect invariant that $newBal = bal - 100$ in the withdraw function is often reported by Daikon.
The reason for this is that the kernel timeslice is often long enough such that the two threads do not interfere with each other
Reporting this to the developer implies that the update in withdraw is atomic
However, Udon, which uses a systematic exploration of the state space produces the correct invariant: newBal ¡ bal
We refer to these invariants over blocks of code as transition invariants. We believe they are particularly useful in analyzing multithreaded programs.

Likely Invariant Generation of Concurrent Programs

- Why Not just use Daikon? [Ernst et al., 2007]
- Naive exploration of the state space often misses the bug

Shared variable

Atomic read

Atomic write

Non-atomic read-modify-write

Concurrent withdraws

```
int balance = 400;
int getBalance() {
  int bal;
  Lock();
  bal = balance;
  Unlock();
  return bal;
}
void setBalance(int bal) {
  Lock();
  balance = bal;
  Unlock();
}
void withdraw() {
  int bal = getBalance();
  newBal = bal - 100;
  setBalance(newBal);
}
int main(void) {
  thread_create(&t1,withdraw);
  thread_create(&t2,withdraw);
  thread_join(t1);
  thread_join(t1);
  assert(balance==200);
}
```


This example shows why using Daikon on concurrent programs is not acurrate
The main reason is that a naive exploration of the state space often misses concurrent behavior
In this program, two threads are concurrently modifying a shared variable balance initialized to 400.
The getBalance function is an atomic read, and the setBalance function is an atomic write
However, the withdraw function is a non-atomic read-modify-write
As a result, when two threads are concurrently withdrawing 100, the final state can be either 300 or 200
Since the bug is often missed, the incorrect invariant that $newBal = bal - 100$ in the withdraw function is often reported by Daikon.
The reason for this is that the kernel timeslice is often long enough such that the two threads do not interfere with each other
Reporting this to the developer implies that the update in withdraw is atomic
However, Udon, which uses a systematic exploration of the state space produces the correct invariant: newBal ¡ bal
We refer to these invariants over blocks of code as transition invariants. We believe they are particularly useful in analyzing multithreaded programs.

Likely Invariant Generation of Concurrent Programs

- Why Not just use Daikon? [Ernst et al., 2007]
- Naive exploration of the state space often misses the bug

```
int balance = 400;
int getBalance() {
  int bal;
  Lock();
  bal = balance;
  Unlock();
  return bal;
}
void setBalance(int bal) {
  Lock();
  balance = bal;
  Unlock();
}
void withdraw() {
  int bal = getBalance();
  newBal = bal - 100;
  setBalance(newBal);
}
int main(void) {
  thread_create(&t1,withdraw);
  thread_create(&t2,withdraw);
  thread_join(t1);
  thread_join(t1);
  assert(balance==200);
}
```

Shared variable

Atomic read

Atomic write

Non-atomic read-modify-write

Concurrent withdraws

Violated: 200 or 300

This example shows why using Daikon on concurrent programs is not acurrate
The main reason is that a naive exploration of the state space often misses concurrent behavior
In this program, two threads are concurrently modifying a shared variable balance initialized to 400.
The getBalance function is an atomic read, and the setBalance function is an atomic write
However, the withdraw function is a non-atomic read-modify-write
As a result, when two threads are concurrently withdrawing 100, the final state can be either 300 or 200
Since the bug is often missed, the incorrect invariant that $newBal = bal - 100$ in the withdraw function is often reported by Daikon.
The reason for this is that the kernel timeslice is often long enough such that the two threads do not interfere with each other
Reporting this to the developer implies that the update in withdraw is atomic
However, Udon, which uses a systematic exploration of the state space produces the correct invariant: newBal ¡ bal
We refer to these invariants over blocks of code as transition invariants. We believe they are particularly useful in analyzing multithreaded programs.

Likely Invariant Generation of Concurrent Programs

- Why Not just use Daikon? [Ernst et al., 2007]
- Naive exploration of the state space often misses the bug
- Incorrect Invariant: $newBal = bal - 100$

```
int balance = 400;
int getBalance() {
  int bal;
  Lock();
  bal = balance;
  Unlock();
  return bal;
}
void setBalance(int bal) {
  Lock();
  balance = bal;
  Unlock();
}
void withdraw() {
  int bal = getBalance();
  newBal = bal - 100;
  setBalance(newBal);
}
int main(void) {
  thread_create(&t1,withdraw);
  thread_create(&t2,withdraw);
  thread_join(t1);
  thread_join(t1);
  assert(balance==200);
}
```

Shared variable

Atomic read

Atomic write

Non-atomic read-modify-write

$newBal \neq bal - 100$

Concurrent withdraws

Violated: 200 or 300

This example shows why using Daikon on concurrent programs is not acurrate
The main reason is that a naive exploration of the state space often misses concurrent behavior
In this program, two threads are concurrently modifying a shared variable balance initialized to 400.
The getBalance function is an atomic read, and the setBalance function is an atomic write
However, the withdraw function is a non-atomic read-modify-write
As a result, when two threads are concurrently withdrawing 100, the final state can be either 300 or 200
Since the bug is often missed, the incorrect invariant that $newBal = bal - 100$ in the withdraw function is often reported by Daikon.
The reason for this is that the kernel timeslice is often long enough such that the two threads do not interfere with each other
Reporting this to the developer implies that the update in withdraw is atomic
However, Udon, which uses a systematic exploration of the state space produces the correct invariant: newBal ¡ bal
We refer to these invariants over blocks of code as transition invariants. We believe they are particularly useful in analyzing multithreaded programs.

Likely Invariant Generation of Concurrent Programs

- Why Not just use Daikon? [Ernst et al., 2007]
- Naive exploration of the state space often misses the bug
- Incorrect Invariant: *newBal* $=$ *bal* $-$ 100
- Udon systematically explores the concurrent state space

Shared variable

Atomic read

Atomic write

Non-atomic read-modify-write

newBal $\neq$ *bal* $-$ 100

Concurrent withdraws

Violated: 200 or 300

```
int balance = 400;
int getBalance() {
  int bal;
  Lock();
  bal = balance;
  Unlock();
  return bal;
}
void setBalance(int bal) {
  Lock();
  balance = bal;
  Unlock();
}
void withdraw() {
  int bal = getBalance();
  newBal = bal - 100;
  setBalance(newBal);
}
int main(void) {
  thread_create(&t1,withdraw);
  thread_create(&t2,withdraw);
  thread_join(t1);
  thread_join(t1);
  assert(balance==200);
}
```


This example shows why using Daikon on concurrent programs is not acurrate
The main reason is that a naive exploration of the state space often misses concurrent behavior
In this program, two threads are concurrently modifying a shared variable balance initialized to 400.
The getBalance function is an atomic read, and the setBalance function is an atomic write
However, the withdraw function is a non-atomic read-modify-write
As a result, when two threads are concurrently withdrawing 100, the final state can be either 300 or 200
Since the bug is often missed, the incorrect invariant that *newBal* $=$ *bal* $-$ 100 in the withdraw function is often reported by Daikon.
The reason for this is that the kernel timeslice is often long enough such that the two threads do not interfere with each other
Reporting this to the developer implies that the update in withdraw is atomic
However, Udon, which uses a systematic exploration of the state space produces the correct invariant: newBal ¡ bal
We refer to these invariants over blocks of code as transition invariants. We believe they are particularly useful in analyzing multithreaded programs.

Likely Invariant Generation of Concurrent Programs

- Why Not just use Daikon? [Ernst et al., 2007]
- Naive exploration of the state space often misses the bug
- Incorrect Invariant: $newBal = bal - 100$
- Udon systematically explores the concurrent state space
- Correct Invariant: $newBal < bal$

```
int balance = 400;
int getBalance() {
  int bal;
  Lock();
  bal = balance;
  Unlock();
  return bal;
}
void setBalance(int bal) {
  Lock();
  balance = bal;
  Unlock();
}
void withdraw() {
  int bal = getBalance();
  newBal = bal - 100;
  setBalance(newBal);
}
int main(void) {
  thread_create(&t1,withdraw);
  thread_create(&t2,withdraw);
  thread_join(t1);
  thread_join(t1);
  assert(balance==200);
}
```

Shared variable
Atomic read
Atomic write
Non-atomic read-modify-write
$newBal < bal$
Concurrent withdraws
Violated: 200 or 300

This example shows why using Daikon on concurrent programs is not acurrate
The main reason is that a naive exploration of the state space often misses concurrent behavior
In this program, two threads are concurrently modifying a shared variable balance initialized to 400.
The getBalance function is an atomic read, and the setBalance function is an atomic write
However, the withdraw function is a non-atomic read-modify-write
As a result, when two threads are concurrently withdrawing 100, the final state can be either 300 or 200
Since the bug is often missed, the incorrect invariant that $newBal = bal - 100$ in the withdraw function is often reported by Daikon.
The reason for this is that the kernel timeslice is often long enough such that the two threads do not interfere with each other
Reporting this to the developer implies that the update in withdraw is atomic
However, Udon, which uses a systematic exploration of the state space produces the correct invariant: newBal ¡ bal
We refer to these invariants over blocks of code as transition invariants. We believe they are particularly useful in analyzing multithreaded programs.

Likely Invariant Generation of Concurrent Programs

- Why Not just use Daikon? [Ernst et al., 2007]
- Naive exploration of the state space often misses the bug
- Incorrect Invariant: *newBal* $=$ *bal* $-$ 100
- Udon systematically explores the concurrent state space
- Correct Invariant: *newBal* $<$ *bal*
- Transition Invariants

Shared variable

Atomic read

Atomic write

Non-atomic read-modify-write

newBal $<$ *bal*

Concurrent withdraws

Violated: 200 or 300

```
int balance = 400;
int getBalance() {
  int bal;
  Lock();
  bal = balance;
  Unlock();
  return bal;
}
void setBalance(int bal) {
  Lock();
  balance = bal;
  Unlock();
}
void withdraw() {
  int bal = getBalance();
  newBal = bal - 100;
  setBalance(newBal);
}
int main(void) {
  thread_create(&t1,withdraw);
  thread_create(&t2,withdraw);
  thread_join(t1);
  thread_join(t1);
  assert(balance==200);
}
```


This example shows why using Daikon on concurrent programs is not acurrate
The main reason is that a naive exploration of the state space often misses concurrent behavior
In this program, two threads are concurrently modifying a shared variable balance initialized to 400.
The getBalance function is an atomic read, and the setBalance function is an atomic write
However, the withdraw function is a non-atomic read-modify-write
As a result, when two threads are concurrently withdrawing 100, the final state can be either 300 or 200
Since the bug is often missed, the incorrect invariant that *newBal* $=$ *bal* $-$ 100 in the withdraw function is often reported by Daikon.
The reason for this is that the kernel timeslice is often long enough such that the two threads do not interfere with each other
Reporting this to the developer implies that the update in withdraw is atomic
However, Udon, which uses a systematic exploration of the state space produces the correct invariant: newBal ¡ bal
We refer to these invariants over blocks of code as transition invariants. We believe they are particularly useful in analyzing multithreaded programs.

Likely Invariant Generation of Concurrent Programs

- Udon: More accurate, minimal overhead

```
int balance = 400;
int getBalance() {
  int bal;
  Lock();
  bal = balance;
  Unlock();
  return bal;
}
void setBalance(int bal) {
  Lock();
  balance = bal;
  Unlock();
}
void withdraw() {
  int bal = getBalance();
  newBal = bal - 100;
  setBalance(newBal);
}
int main(void) {
  thread_create(&t1,withdraw);
  thread_create(&t2,withdraw);
  thread_join(t1);
  thread_join(t1);
  assert(balance==200);
}
```


Examining the concrete results of our method on this same program shows that Udon is more acurrate and scalable
Running this program through Daikon results in 78 invariants being generated. 15 of them, about twenty percent, are incorrect
Udon however generates 121 invariants with only 2 being incorrect

Likely Invariant Generation of Concurrent Programs

- Udon: More accurate, minimal overhead
- Daikon: 78 Invariants (15 incorrect)

```
int balance = 400;
int getBalance() {
  int bal;
  Lock();
  bal = balance;
  Unlock();
  return bal;
}
void setBalance(int bal) {
  Lock();
  balance = bal;
  Unlock();
}
void withdraw() {
  int bal = getBalance();
  newBal = bal - 100;
  setBalance(newBal);
}
int main(void) {
  thread_create(&t1,withdraw);
  thread_create(&t2,withdraw);
  thread_join(t1);
  thread_join(t1);
  assert(balance==200);
}
```


Examining the concrete results of our method on this same program shows that Udon is more acurrate and scalable
Running this program through Daikon results in 78 invariants being generated. 15 of them, about twenty percent, are incorrect
Udon however generates 121 invariants with only 2 being incorrect

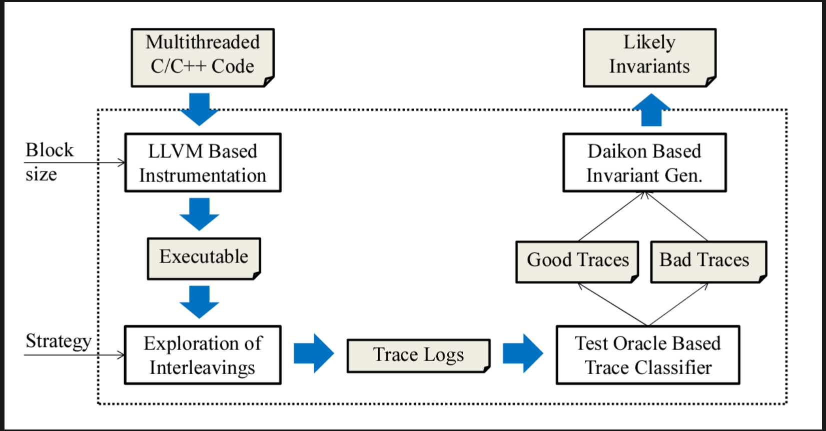
Likely Invariant Generation of Concurrent Programs

- Udon: More accurate, minimal overhead
- Daikon: 78 Invariants (15 incorrect)
- Udon: 121 Invariants (2 incorrect)

```
int balance = 400;
int getBalance() {
  int bal;
  Lock();
  bal = balance;
  Unlock();
  return bal;
}
void setBalance(int bal) {
  Lock();
  balance = bal;
  Unlock();
}
void withdraw() {
  int bal = getBalance();
  newBal = bal - 100;
  setBalance(newBal);
}
int main(void) {
  thread_create(&t1,withdraw);
  thread_create(&t2,withdraw);
  thread_join(t1);
  thread_join(t1);
  assert(balance==200);
}
```


Examining the concrete results of our method on this same program shows that Udon is more acurrate and scalable
Running this program through Daikon results in 78 invariants being generated. 15 of them, about twenty percent, are incorrect
Udon however generates 121 invariants with only 2 being incorrect

Overview



Here is a high level overview of our method
We take as input a multithreaded program, a block size, and a search strategy
The block size is the number of lines of code which we generate transition invariants over
The strategy is the systematic exploration strategy used by the stateless model checking
We use the LLVM compiler framework to instrument the code for dynamic analysis
Our tool then explores the concurrent state space to generate a log of trace data for each run
Then, our tool allows for both passing and failing runs to be seperated in different categories
This allows for invariants to be generated showing the difference between the correct and incorrect behavior
Finally, we pass the traces to a previous invariant generator from the Daikon project.

Overview

Next, I will present our experimental results

Setup

- Compared Udon to Daikon
- For all the invariants generated, we manually checked if they were correct
- Does Daikon work on concurrent programs?
- Does Udon work on concurrent programs?
- Scalability?

Professor Frink: Wikipedia

We compared our approach to the existing state-of-the-art tool named Daikon
We tested on 19 different concurrent programs
For each test, we manually checked if any of the invariants generated by Daikon or Udon were incorrect
We wanted to answer the following questions:
Does the prior art, Daikon, generate correct invariants for concurrent programs?
Can our new Udon approach handle concurrent programs?
Can our new method scale?

Average Results

Daikon	Daikon*	Udon

We tested Daikon in two different ways: first, we allowed it only to execute the concurrent program once and second we allowed it to run as many times as Udon
This slides shows the average results of all tests
In both cases, it used a thread schedule selected by the operating system
We refer to these two methods as daikon and daikon star
First, we can see that running Daikon once or many times results in around the same number of invariants being generated
Udon, on the other hand, finds many more invariants
However, in both cases, Daikon produces many incorrect invariants
Udon, on the other hand, produced only one incorrect invariant on average
These incorrect invariants were usually caused by the heuristic based search strategy used in Udon
Additionally, because of our static instrumentation using LLVM we have a reduction in runtime compared to Daikon.
This is because Daikon dynamically instruments the binary at runtime. This needs to be repeated on each repeated run.

Average Results

Daikon	Daikon*	Udon
220 Invariants	234 Invariants	332 Invariants

We tested Daikon in two different ways: first, we allowed it only to execute the concurrent program once and second we allowed it to run as many times as Udon
This slides shows the average results of all tests
In both cases, it used a thread schedule selected by the operating system
We refer to these two methods as daikon and daikon star
First, we can see that running Daikon once or many times results in around the same number of invariants being generated
Udon, on the other hand, finds many more invariants
However, in both cases, Daikon produces many incorrect invariants
Udon, on the other hand, produced only one incorrect invariant on average
These incorrect invariants were usually caused by the heuristic based search strategy used in Udon
Additionally, because of our static instrumentation using LLVM we have a reduction in runtime compared to Daikon.
This is because Daikon dynamically instruments the binary at runtime. This needs to be repeated on each repeated run.

Average Results

Daikon	Daikon*	Udon
220 Invariants	234 Invariants	332 Invariants
16 Incorrect	15 Incorrect	1 Incorrect

We tested Daikon in two different ways: first, we allowed it only to execute the concurrent program once and second we allowed it to run as many times as Udon
This slides shows the average results of all tests
In both cases, it used a thread schedule selected by the operating system
We refer to these two methods as daikon and daikon star
First, we can see that running Daikon once or many times results in around the same number of invariants being generated
Udon, on the other hand, finds many more invariants
However, in both cases, Daikon produces many incorrect invariants
Udon, on the other hand, produced only one incorrect invariant on average
These incorrect invariants were usually caused by the heuristic based search strategy used in Udon
Additionally, because of our static instrumentation using LLVM we have a reduction in runtime compared to Daikon.
This is because Daikon dynamically instruments the binary at runtime. This needs to be repeated on each repeated run.

Average Results

Daikon	Daikon*	Udon
220 Invariants	234 Invariants	332 Invariants
16 Incorrect	15 Incorrect	1 Incorrect
2.8 s	42.9 s	8.6 s

We tested Daikon in two different ways: first, we allowed it only to execute the concurrent program once and second we allowed it to run as many times as Udon
This slides shows the average results of all tests
In both cases, it used a thread schedule selected by the operating system
We refer to these two methods as daikon and daikon star
First, we can see that running Daikon once or many times results in around the same number of invariants being generated
Udon, on the other hand, finds many more invariants
However, in both cases, Daikon produces many incorrect invariants
Udon, on the other hand, produced only one incorrect invariant on average
These incorrect invariants were usually caused by the heuristic based search strategy used in Udon
Additionally, because of our static instrumentation using LLVM we have a reduction in runtime compared to Daikon.
This is because Daikon dynamically instruments the binary at runtime. This needs to be repeated on each repeated run.

Average Results

Daikon	Daikon*	Udon
220 Invariants	234 Invariants	332 Invariants
16 Incorrect	15 Incorrect	1 Incorrect
2.8 s	42.9 s	8.6 s

Faster, More Accurate

We tested Daikon in two different ways: first, we allowed it only to execute the concurrent program once and second we allowed it to run as many times as Udon
This slides shows the average results of all tests
In both cases, it used a thread schedule selected by the operating system
We refer to these two methods as daikon and daikon star
First, we can see that running Daikon once or many times results in around the same number of invariants being generated
Udon, on the other hand, finds many more invariants
However, in both cases, Daikon produces many incorrect invariants
Udon, on the other hand, produced only one incorrect invariant on average
These incorrect invariants were usually caused by the heuristic based search strategy used in Udon
Additionally, because of our static instrumentation using LLVM we have a reduction in runtime compared to Daikon.
This is because Daikon dynamically instruments the binary at runtime. This needs to be repeated on each repeated run.

Average Results

DPOR	PCB	HaPSet

The previous results all used the happy set search strategy
We compared the effect of different search strategies on the quality of generated invariants
DPOR, while guaranteed to explore all concurrent behavior does not scale as well as HaPSet or PCB
As expected, DPOR explores many more executions than PCB and HapSet
Of them all, HaPSet explores the smallest number of executions, and thus has the lowest runtime
Interestingly, HaPSet also generates fewer incorrect invariants compared to PCB
This suggests that HaPSet provides better coverage of the concurrent behavior than PCB
HaPSet seems to provide a good trade off between scalability and accuracy
Because of this, the default search strategy in Udon is HaPSet

Average Results

DPOR	PCB	HaPSet
6187 runs	115 runs	42 runs

The previous results all used the happy set search strategy
We compared the effect of different search strategies on the quality of generated invariants
DPOR, while guaranteed to explore all concurrent behavior does not scale as well as HaPSet or PCB
As expected, DPOR explores many more executions than PCB and HapSet
Of them all, HaPSet explores the smallest number of executions, and thus has the lowest runtime
Interestingly, HaPSet also generates fewer incorrect invariants compared to PCB
This suggests that HaPSet provides better coverage of the concurrent behavior than PCB
HaPSet seems to provide a good trade off between scalability and accuracy
Because of this, the default search strategy in Udon is HaPSet

Average Results

DPOR	PCB	HaPSet
6187 runs	115 runs	42 runs
161.2 s	11.31 s	8.75 s

The previous results all used the happy set search strategy
We compared the effect of different search strategies on the quality of generated invariants
DPOR, while guaranteed to explore all concurrent behavior does not scale as well as HaPSet or PCB
As expected, DPOR explores many more executions than PCB and HapSet
Of them all, HaPSet explores the smallest number of executions, and thus has the lowest runtime
Interestingly, HaPSet also generates fewer incorrect invariants compared to PCB
This suggests that HaPSet provides better coverage of the concurrent behavior than PCB
HaPSet seems to provide a good trade off between scalability and accuracy
Because of this, the default search strategy in Udon is HaPSet

Average Results

DPOR	PCB	HaPSet
6187 runs	115 runs	42 runs
161.2 s	11.31 s	8.75 s
0 incorrect	4 incorrect	1 incorrect

The previous results all used the happy set search strategy
We compared the effect of different search strategies on the quality of generated invariants
DPOR, while guaranteed to explore all concurrent behavior does not scale as well as HaPSet or PCB
As expected, DPOR explores many more executions than PCB and HapSet
Of them all, HaPSet explores the smallest number of executions, and thus has the lowest runtime
Interestingly, HaPSet also generates fewer incorrect invariants compared to PCB
This suggests that HaPSet provides better coverage of the concurrent behavior than PCB
HaPSet seems to provide a good trade off between scalability and accuracy
Because of this, the default search strategy in Udon is HaPSet

Conclusion

- Udon: the first (robust) dynamic invariant generator for multithreaded programs
- LLVM front end for Daikon's invariant generator
- Accurate: produces few incorrect invariants
- Accurate: produces more correct invariants
- Scalable: minimal overhead

japancentre.com

In conclusion, we presented Udon, the first dynamic invariant generator for multithreaded programs
We created an LLVM front end for Daikon's invariant generator
Combined stateless model checking with dynamic invariant generation
We showed our method is accurate and scalable
With that, I'll take any questions.

Conclusion

- Udon: the first (robust) dynamic invariant generator for multithreaded programs
- LLVM front end for Daikon's invariant generator
- Accurate: produces few incorrect invariants
- Accurate: produces more correct invariants
- Scalable: minimal overhead

Questions?

japancentre.com

In conclusion, we presented Udon, the first dynamic invariant generator for multithreaded programs
We created an LLVM front end for Daikon's invariant generator
Combined stateless model checking with dynamic invariant generation
We showed our method is accurate and scalable
With that, I'll take any questions.

References

- [Ernst et al., 2007]: *The Daikon system for dynamic detection of likely invariants*, Michael D. Ernst et al. Science of Computer Programming, Dec. 2007,
- [Flanagan and Godefroid, 2005]: *Dynamic partial-order reduction for model checking software*, Cormac Flanagan, and Patrice Godefroid. POPL 2005
- [Musuvathi and Qadeer, 2007]: *Iterative Context Bounding for Systematic Testing of Multithreaded Programs*, Madan Musuvathi, and Shaz Qadeer. PLDI 2007
- [Godefroid, 1997]: *Model Checking for Programming Languages using VeriSoft* Godefroid. 1997
- [Wang et al., 2011]: *Coverage Guided Systematic Concurrency Testing*, Chao Wang et al. ICSE 2011

Results

	Number of Invariants			Incorrect Invariants			Number of Runs			Run Time (s)		
Name	Daikon	Daikon*	Udon	Daikon	Daikon*	Udon	Daikon	Daikon*	Udon	Daikon	Daikon*	Udon
Sync01_Safe	15	15	15	1	1	0	1	4	4	1.6	2.8	4.3
FibBenchSafe	24	24	17	10	10	0	1	6	6	2.9	3.4	4.2
Lazy01Safe	20	22	22	7	4	0	1	9	9	2.6	4.3	4.8
Stateful01_Safe	21	21	21	6	6	3	1	4	4	1.4	2.7	3.9
DekkerSafe	39	44	52	29	24	8	1	53	53	1.7	19.5	4.8
LamportSafe	48	59	76	36	44	2	1	58	58	5.0	21.5	5.3
PetersonSafe	39	39	57	29	29	0	1	46	46	1.7	17.3	4.7
TimeVarMutex	27	27	24	9	9	0	1	3	3	1.6	2.2	3.7
Szymanski	30	32	35	25	24	0	1	111	111	1.6	39.4	5.2
IncTrue	15	19	30	3	3	0	1	19	19	2.2	7.7	3.9
IncCas	14	19	38	3	3	0	1	9	9	2.7	3.7	3.6
IncDec	95	122	205	57	53	0	1	39	39	3.4	15.7	6.3
IncDecCas	49	49	73	38	38	1	1	8	8	2.9	4.3	4.8
Reorder	44	54	95	8	6	0	1	29	29	2.5	10.6	7.4
AccountBad	74	78	121	22	15	2	1	9	9	3.8	5.4	6.9
Pfscan	670	798	840	9	9	0	1	20	20	3.0	13.8	8.9
nbds-hashtable	1123	1194	2064	2	2	0	1	74	74	5.4	119.7	39.0
nbds-skiplist01	1053	1055	1370	1	1	0	1	161	161	4.4	287.6	26.2
nbds-list_idx01	773	773	1143	1	1	0	1	132	132	4.6	235.2	17.0
Average	220	234	332	16	15	1	1	42	42	2.8	42.9	8.6

Results

Name	Number of Runs			Run Time (s)			Incorrect Invariants		
	HaPSet	PCB	DPOR	HaPSet	PCB	DPOR	HaPSet	PCB	DPOR
Sync01_Safe	4	14	7	4.3	4.3	4.7	0	0	0
FibBenchSafe	6	33	17K	4.2	4.3	139.1	0	0	0
Lazy01Safe	9	49	40	4.8	5.3	5.5	0	1	0
Stateful01_Safe	4	13	12	3.9	4.2	4.2	3	1	0
DekkerSafe	53	13	3896	4.8	4.4	37.6	8	16	0
LamportSafe	58	19	392	5.3	4.6	9.4	2	9	0
PetersonSafe	46	13	730	4.7	4.4	12.4	0	0	0
TimeVarMutex	3	17	4	3.7	4.0	4.2	0	4	0
Szymanski	111	21	5980	5.2	4.2	50.3	0	11	0
IncTrue	19	17	212	3.9	3.8	6.1	0	2	0
IncCas	9	19	33	3.6	4.4	5.0	0	0	0
IncDec	39	52	484	6.3	6.7	12.0	0	0	0
IncDecCas	8	22	30	4.8	5.0	5.6	1	8	0
Reorder	29	506	19K	7.4	12.5	234.9	0	2	0
AccountBad	9	53	40	6.9	7.4	7.8	2	19	0
Pfscan	20	100	56K	8.9	11.8	2263	0	0	0
nbds-hashtable	74	879	✗	39.0	86.1	✗	0	0	✗
nbds-skiplist01	161	249	10K	26.2	20.6	217.0	0	0	0
nbds-list_idx01	132	85	1498	17.0	16.0	44.5	0	0	0
Average:	42	115	6187	8.75	11.31	161.2	1	4	0