Dynamic Generation of Likely Invariants for Multithreaded Programs

ICSE 2015

Markus Kusano  Arijit Chattopadhyay  Chao Wang

Virginia Tech, USA

May 21, 2015
Introduction

“\text{It would be easier to rewrite this entire thing than to understand it}”

\begin{verbatim}
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;
}
\end{verbatim}

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.

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

    x2 = number * 0.5F;
    i = * ( long * ) &y;
    // evil floating point bit hacking
    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;
}
```

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

```c
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.

Knowing the runtime behavior of a program is essential but difficult.

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

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

Knowing the runtime behavior of a program is essential but difficult

• 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

```cpp
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;
    }
  }
};
```

The problem is made even more difficult for concurrent programs. Complex interactions between threads are difficult to reason about. Here's 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 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

```cpp
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 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 used by a single producer? If it is shared, then this non-atomic update is a bug.

Pointers 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.
Concurrent programs make things even worse

Are our assumptions correct?

Invariants can provide answers to these questions

The problem is made even more difficult for concurrent programs
Complex interactions between threads are difficult to reason about
Here’s some code, written by a fairly experienced C++ programmer, implementing a lock-free queue

```cpp
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) {
            Node* tmp = first;
            first = first->next;
            delete tmp;
        }
    }
};
```

Dr. Dobbs, Herb Sutter, Lock-free Queue

Is there only one producer?

Non-atomic modification
Introduction

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

The problem is made even more difficult for concurrent programs. Complex interactions between threads are difficult to reason about. Here's some code, written by a fairly experienced C++ programmer, implementing a lock-free queue:

```cpp
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 = last->next = new Node(t);
        last = last->next; // publish it
        while (first != divider) { // trim
            Node* tmp = first;
            first = first->next;
            delete tmp;
        }
    }
};
```

Again, the programmer makes a few assumptions:

- The pointer to the beginning of the list is not-atomic; are we sure it is only 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.
Next, I will go over both true and likely invariants.
Invariants

An invariant is a truth condition at a particular location.

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

```c
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.
An invariant is a truth condition at a particular location.

It is true on all possible paths of the program.

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:

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

One useful invariant about this function is that its output is always greater than its input. This is a nice compact summary of the function’s behavior.

Next, consider this function which indirectly increments a value through a pointer:

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

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.
An invariant is a truth condition at a particular location

It is true on all possible paths of the program

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:

```c
int inc(int i) {
    ret = i + 1;  // ret > x
    return ret;
}
```

One useful invariant about this function is that its output is always greater than its input. This is a nice compact summary of the function’s behavior.

Next, consider this function which indirectly increments a value through a pointer:

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

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.
An invariant is a truth condition at a particular location.

It is true on all possible paths of the program.

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 function’s 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.
An invariant is a truth condition at a particular location.

It is true on all possible paths of the program.

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:

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

One useful invariant about this function is that its output is always greater than its input. This is a nice compact summary of the function’s behavior.

Next, consider this function which indirectly increments a value through a pointer:

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

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

```c
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

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

```c
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.
  - \texttt{inc(3)}.
  - \texttt{inc_ptr(0x0AB7FC5B)}

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.

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

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

```c
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.
A likely invariant is a truth condition at a particular location. It is true on some possible paths of the program. For example:

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

```c
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:

- `inc(3)`, `inc_ptr(0x0AB7FC5B)`
- `inc(7)`, `inc_ptr(0x0AB7F194)`

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)`

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]

```c
int inc(int i) {
    ret = i + 1; [ret = (4, 8)]
    return ret;
}
```

```c
void inc_ptr(int *p) {
    ret = p) + 1; [p != 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.
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

```c
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 executions are chosen.
Stateless Model Checking of Concurrent Programs

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

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 executions are chosen.

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

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 executions are chosen.

```c
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 executions are chosen.
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 executions are chosen.

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

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]

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.
Next, I will discuss our new tool, Udon.
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(t2);
  assert(balace==200);
}
```

This example shows why using Daikon on concurrent programs is not accurate. 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 \leq bal`.

We refer to these invariants over blocks of code as transition invariants. We believe they are particularly useful in analyzing multithreaded programs.
This example shows why using Daikon on concurrent programs is not accurate. 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 \(\text{newBal} = \text{bal} - 100\) 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: \(\text{newBal} \leq \text{bal}\).

We refer to these invariants over blocks of code as transition invariants. We believe they are particularly useful in analyzing multithreaded programs.
Why Not just use Daikon?
[Ernst et al., 2007]

Naive exploration of the state space often misses the bug

---

```c
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 accurate.
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.
Why Not just use Daikon?
[Ernst et al., 2007]

Naive exploration of the state space often misses the bug

This example shows why using Daikon on concurrent programs is not accurate. 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 \( \text{newBal} = \text{bal} - 100 \) 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: \( \text{newBal} < \text{bal} \)

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

This example shows why using Daikon on concurrent programs is not accurate. 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.
Why Not just use Daikon?  
[Ernst et al., 2007]  
Naive exploration of the state space often misses the bug

Incorrect Invariant:  
\[
\text{newBal} = \text{bal} - 100
\]

Correct Invariant:  
\[
\text{newBal} < \text{bal}
\]

This example shows why using Daikon on concurrent programs is not accurate.  
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.
Why Not just use Daikon? \cite{Ernst2007} 

Naive exploration of the state space often misses the bug

Incorrect Invariant:
\[ \text{newBal} = \text{bal} - 100 \]

Concurrent withdraws

This example shows why using Daikon on concurrent programs is not accurate. 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 \( \text{newBal} = \text{bal} - 100 \) 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: \( \text{newBal} \leq \text{bal} \)

We refer to these invariants over blocks of code as transition invariants. We believe they are particularly useful in analyzing multithreaded programs.
Why Not just use Daikon?
[Ernst et al., 2007]

Naive exploration of the state space often misses the bug

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 \( \text{newBal} = \text{bal} - 100 \)
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: \( \text{newBal} < \text{bal} \)
We refer to these invariants over blocks of code as transition invariants. We believe they are particularly useful in analyzing multithreaded programs.
Why Not just use Daikon?
[Ernst et al., 2007]

Naive exploration of the state space often misses the bug

Incorrect Invariant:
\[
\text{newBal} = \text{bal} - 100
\]

Transition Invariants

```
int balance = 400;

int getBalance () {
    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

```
newBal \neq bal - 100
```

This example shows why using Daikon on concurrent programs is not accurate. 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 \(\text{newBal} = \text{bal} - 100\) 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: \(\text{newBal} < \text{bal}\).

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

This example shows why using Daikon on concurrent programs is not accurate. 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: $newBal \leq bal$

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

This example shows why using Daikon on concurrent programs is not accurate. 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.
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

This example shows why using Daikon on concurrent programs is not accurate. 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

```c
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(t2);
    assert(balance == 200);
}
```

Examining the concrete results of our method on this same program shows that Udon is more accurate 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.
Examining the concrete results of our method on this same program shows that Udon is more accurate 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)

```c
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(t2);
    assert(balance==200);
}
```

Examining the concrete results of our method on this same program shows that Udon is more accurate 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.
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 separated 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.
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?

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?
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 slide 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.
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 slide 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.
<table>
<thead>
<tr>
<th></th>
<th>Daikon</th>
<th>Daikon*</th>
<th>Udon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invariants</td>
<td>220</td>
<td>234</td>
<td>332</td>
</tr>
<tr>
<td>Incorrect</td>
<td>16</td>
<td>15</td>
<td>1</td>
</tr>
</tbody>
</table>

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 slide 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.
<table>
<thead>
<tr>
<th></th>
<th>Daikon</th>
<th>Daikon*</th>
<th>Udon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invariants</td>
<td>220</td>
<td>234</td>
<td>332</td>
</tr>
<tr>
<td>Incorrect</td>
<td>16</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>Time</td>
<td>2.8 s</td>
<td>42.9 s</td>
<td>8.6 s</td>
</tr>
</tbody>
</table>

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 slide 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

<table>
<thead>
<tr>
<th></th>
<th>Daikon</th>
<th>Daikon*</th>
<th>Udon</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>220 Invariants</td>
<td>234 Invariants</td>
<td>332 Invariants</td>
</tr>
<tr>
<td></td>
<td>16 Incorrect</td>
<td>15 Incorrect</td>
<td>1 Incorrect</td>
</tr>
<tr>
<td></td>
<td>2.8 s</td>
<td>42.9 s</td>
<td>8.6 s</td>
</tr>
</tbody>
</table>

**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 slide 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.
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.
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.
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.
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.
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.
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

- [Wang et al., 2011]: *Coverage Guided Systematic Concurrency Testing*, Chao Wang et al. ICSE 2011
<table>
<thead>
<tr>
<th>Name</th>
<th>Number of Invariants</th>
<th>Incorrect Invariants</th>
<th>Number of Runs</th>
<th>Run Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Daikon</td>
<td>Daikon*</td>
<td>Udon</td>
<td>Daikon</td>
</tr>
<tr>
<td>Sync01Safe</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>FibBenchSafe</td>
<td>24</td>
<td>24</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>Lazy01Safe</td>
<td>20</td>
<td>20</td>
<td>22</td>
<td>7</td>
</tr>
<tr>
<td>Stateful01Safe</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>6</td>
</tr>
<tr>
<td>DekkerSafe</td>
<td>39</td>
<td>46</td>
<td>52</td>
<td>29</td>
</tr>
<tr>
<td>LamportSafe</td>
<td>48</td>
<td>59</td>
<td>70</td>
<td>36</td>
</tr>
<tr>
<td>PetersonSafe</td>
<td>39</td>
<td>39</td>
<td>57</td>
<td>29</td>
</tr>
<tr>
<td>TimeVarMutex</td>
<td>27</td>
<td>27</td>
<td>24</td>
<td>9</td>
</tr>
<tr>
<td>Synchronizer</td>
<td>30</td>
<td>32</td>
<td>35</td>
<td>25</td>
</tr>
<tr>
<td>IncTrue</td>
<td>15</td>
<td>19</td>
<td>30</td>
<td>3</td>
</tr>
<tr>
<td>IncCas</td>
<td>14</td>
<td>19</td>
<td>38</td>
<td>3</td>
</tr>
<tr>
<td>IncDec</td>
<td>95</td>
<td>122</td>
<td>206</td>
<td>57</td>
</tr>
<tr>
<td>IncDecCas</td>
<td>49</td>
<td>48</td>
<td>71</td>
<td>38</td>
</tr>
<tr>
<td>Reorder</td>
<td>44</td>
<td>54</td>
<td>95</td>
<td>8</td>
</tr>
<tr>
<td>AccountBad</td>
<td>74</td>
<td>78</td>
<td>131</td>
<td>22</td>
</tr>
<tr>
<td>Pthreads</td>
<td>870</td>
<td>781</td>
<td>840</td>
<td>9</td>
</tr>
<tr>
<td>nbsd-hashable</td>
<td>1123</td>
<td>1194</td>
<td>2064</td>
<td>2</td>
</tr>
<tr>
<td>nbsd-skiplist</td>
<td>1053</td>
<td>1055</td>
<td>1370</td>
<td>1</td>
</tr>
<tr>
<td>nbsd-list</td>
<td>773</td>
<td>773</td>
<td>1143</td>
<td>1</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>220</td>
<td>234</td>
<td>332</td>
<td>18</td>
</tr>
</tbody>
</table>
## Results

<table>
<thead>
<tr>
<th>Name</th>
<th>Number of Runs</th>
<th>Run Time (s)</th>
<th>Incorrect Invariants</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HaPSet</td>
<td>PCB</td>
<td>DPOR</td>
</tr>
<tr>
<td>Sync01_Safe</td>
<td>4</td>
<td>14</td>
<td>7</td>
</tr>
<tr>
<td>FibBenchSafe</td>
<td>6</td>
<td>33</td>
<td>17K</td>
</tr>
<tr>
<td>Lazy01Safe</td>
<td>9</td>
<td>49</td>
<td>40</td>
</tr>
<tr>
<td>Stateful01_Safe</td>
<td>4</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>DekkerSafe</td>
<td>53</td>
<td>13</td>
<td>3896</td>
</tr>
<tr>
<td>LamportSafe</td>
<td>58</td>
<td>19</td>
<td>392</td>
</tr>
<tr>
<td>PetersonSafe</td>
<td>46</td>
<td>13</td>
<td>730</td>
</tr>
<tr>
<td>TimeVarMutex</td>
<td>3</td>
<td>17</td>
<td>4</td>
</tr>
<tr>
<td>Szymanski</td>
<td>111</td>
<td>21</td>
<td>5980</td>
</tr>
<tr>
<td>IncTrue</td>
<td>19</td>
<td>17</td>
<td>212</td>
</tr>
<tr>
<td>IncCas</td>
<td>9</td>
<td>19</td>
<td>33</td>
</tr>
<tr>
<td>IncDec</td>
<td>39</td>
<td>52</td>
<td>484</td>
</tr>
<tr>
<td>IncDecCas</td>
<td>8</td>
<td>22</td>
<td>30</td>
</tr>
<tr>
<td>Reorder</td>
<td>29</td>
<td>506</td>
<td>19K</td>
</tr>
<tr>
<td>AccountBad</td>
<td>9</td>
<td>53</td>
<td>40</td>
</tr>
<tr>
<td>Pfscan</td>
<td>20</td>
<td>100</td>
<td>56K</td>
</tr>
<tr>
<td>nbds-hashable</td>
<td>74</td>
<td>879</td>
<td>X</td>
</tr>
<tr>
<td>nbds-skiplist01</td>
<td>161</td>
<td>249</td>
<td>10K</td>
</tr>
<tr>
<td>nbds-list_idx01</td>
<td>132</td>
<td>85</td>
<td>1498</td>
</tr>
<tr>
<td><strong>Average:</strong></td>
<td>42</td>
<td>115</td>
<td>6187</td>
</tr>
</tbody>
</table>