Assertion Guided Abstraction: A Cooperative Optimization For Dynamic Partial Order Reduction

Markus Kusano  Chao Wang

Virginia Tech

September 17, 2014
There is a crisis in the development of multithreaded programs.

First, on the design side, due to their astronomical complexity, it is easy for the developer to insert Heisenbugs into the program.

For example, data races and deadlocks are difficult to reproduce bugs which may rarely occur during traditional software testing. These bugs also often cause catastrophic system failure.

The main cause of these bugs is that it is difficult for a human to reason about the complexity of a multithreaded program.

As a result, it would be nice if we had tools to help developers write correct multithreaded programs.
The Problem

Design

Multithreaded Program

- There is a crisis in the development of multithreaded programs.
- First, on the design side, due to their astronomical complexity, it is easy for the developer to insert Heisenbugs into the program.
- For example, data races and deadlocks are difficult to reproduce bugs which may rarely occur during traditional software testing. These bugs also often cause catastrophic system failure.
- The main cause of these bugs is that it is difficult for a human to reason about the complexity of a multithreaded program.
- As a result, it would be nice if we had tools to help developers write correct multithreaded programs.
The Problem

Design

Multithreaded Program

Analysis

• However, concurrent program analysis suffers from the same problem of interleaving explosion. There are far too many interleavings for even a computer to reason about.

• This leaves us in a difficult situation: it is hard to write multithreaded programs and it is hard to analyze multithreaded programs.

• This talk focuses on an optimization for multithreaded program analysis.
However, concurrent program analysis suffers from the same problem of interleaving explosion. There are far too many interleavings for even a computer to reason about.

This leaves us in a difficult situation: it is hard to write multithreaded programs and it is hard to analyze multithreaded programs.

This talk focuses on an optimization for multithreaded program analysis.
An operating system's scheduler is *non-deterministic*

Sometimes bugs don't manifest for a long time

How many schedules are there for a program with:

1. Two threads,
2. Each thread executing 100 instructions

- The root cause of the difficulty in both the design and analysis of multithreaded programs is the non-determinism in the thread scheduler.
- Depending on the schedule of each thread the program may produce different results which may uncover hidden bugs.
- Consider the following: a simple program with two threads each executing only 100 instructions.
- Naively enumerating each thread schedule would require us to test just under $10^{60}$ thread schedules.
The Problem: Non-deterministic Scheduler

- An operating system’s scheduler is *non-deterministic*
- Sometimes bugs don’t manifest for a long time
- How many schedules are there for a program with:
  1. Two threads,
  2. Each thread executing 100 instructions

\[
90,548,514,656,103,281,165,404,177,077,484,163,874,504,589,675, \approx 10^{60}
\]

Motivation

- The root cause of the difficulty in both the design and analysis of multithreaded programs is the non-determinism in the thread scheduler.
- Depending on on the schedule of each thread the program may produce different results which may uncover hidden bugs.
- Consider the following: a simple program with two threads each executing only 100 instructions.
- Naively enumerating each thread schedule would require us to test just under $10^{60}$ thread schedules.
The Problem: Program Analysis

Fact: Multithreaded bugs are a real pest

Solutions:
1. Dynamic analysis [God96, FG05, AAJS14]
2. Static analysis [CE82, CGP99, FK12]
We present a static–dynamic concurrent program analysis
Supplement static analysis with runtime information
We prove our method is sound
Experimentally, we show our method scales to verify previously intractable programs

This brings us to our contribution.
We present a cooperative static and dynamic analysis method.
We allow the static and dynamic analyses to communicate with each other.
Our goal is to get the best of both worlds from both analyses.
We prove our method is sound in that it will not miss any bugs
And, we show using experiments that our method can verify some intractable programs
Next I will go over a comparison of concurrent static and dynamic program analysis techniques.
Stateless model checking + partial order reduction addresses two issues:

1. Too many states to hold in memory
2. Often schedules are equivalent

Dynamic analysis is accurate

But, still doesn’t scale

Too many thread schedules

- One current state-of-the-art dynamic method to verify concurrent programs is stateless model checking coupled with partial order reduction.

- This method addresses two issues: first, often there are too many states in a concurrent program to store in memory, and second often many of the thread schedules are equivalent.

- Dynamic analysis in general has the benefit that it is accurate. Since we are actually running the program, we will never have any false positives.

- However, dynamic analysis still is not perfect. It still suffers from the interleaving explosion problem. There simply are too many thread schedules.
Static Concurrent Program Analysis

- Concurrent control+data flow graphs
- Static analysis has foresight
- Too difficult to simultaneously reason about data and control flow
- Over/under approximate program behavior

- On the other hand, a purely static method at verifying multithreaded programs is to use concurrent control and data flow graphs.
- Static analysis has the benefit of having foresight.
- Consider the control flow graph on the right: static analysis can simultaneously view all possible branches of the program. Dynamic analysis, on the other hand, can only view the executed branch.
- While the extra information is often useful, it also comes as a burden: it is often too difficult to simultaneously reason about all control and data flow information.
- As a result, static analyses often over- or under-approximate program behavior
Current State of Concurrent Program Analysis

- Dynamic analysis doesn’t scale
- Static analysis isn’t accurate

This leaves us at the current and sad situation for concurrent program analysis: dynamic analysis doesn’t scale and static analysis isn’t accurate.

Our contribution is to combine the two to get the best of both worlds.
Next, I will provide a brief background to Dynamic Partial Order Reduction.
Dynamic Partial Order Reduction

- Tests one execution of each equivalence class
- Two sequences are equivalent if they can be obtained by permuting adjacent independent events

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

Possible Executions:
1. \(x = 1; y = 1; y = 2\)
2. \(x = 1; y = 2; y = 1\)
3. \(y = 2; x = 1; y = 1\)

- At the center of dynamic partial order reduction is the notion of dependence between events in the program.
- Based on the definition of dependence, different sequences of events can be grouped into equivalence classes.
- After grouping the events into equivalence classes, DPOR simply tests one representative sequence from each equivalence class.
- Consider the following program. There are three possible thread schedules:
  - First: thread one executes until completion followed by thread two
  - Second: Thread one executes one statement, followed by thread two, and then thread one executes
  - And, finally: thread two executes first followed by thread one
- Dynamic Partial Order Reduction tries to answer the following question: Are any of these sequences redundant?
Dynamic Partial Order Reduction

Tests one execution of each *equivalence class*

Two sequences are equivalent if they can be obtained by permuting adjacent *independent* events

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

Possible Executions:

1. x = 1; y = 1; y = 2
2. x = 1; y = 2; y = 1
3. y = 2; x = 1; y = 1

At the center of dynamic partial order reduction is the notion of dependence between events in the program.

Based on the definition of dependence, different sequences of events can be grouped into equivalence classes.

After grouping the events into equivalence classes, DPOR simply tests one representative sequence from each equivalence class.

Consider the following program. There are three possible thread schedules:

- First: thread one executes until completion followed by thread two
- Second: Thread one executes one statement, followed by thread two, and then thread one executes
- And, finally: thread two executes first followed by thread one

Dynamic Partial Order Reduction tries to answer the following question: Are any of these sequences redundant?
Dynamic Partial Order Reduction

- Tests one execution of each equivalence class
- Two sequences are equivalent if they can be obtained by permuting adjacent independent events

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

Possible Executions:
1. `x = 1; y = 1; y = 2`
2. `x = 1; y = 2; y = 1`
3. `y = 2; x = 1; y = 1`

- At the center of dynamic partial order reduction is the notion of dependence between events in the program.
- Based on the definition of dependence, different sequences of events can be grouped into equivalence classes.
- After grouping the events into equivalence classes, DPOR simply tests one representative sequence from each equivalence class.
- Consider the following program. There are three possible thread schedules:
  - First: thread one executes until completion followed by thread two
  - Second: Thread one executes one statement, followed by thread two, and then thread one executes
  - And, finally: thread two executes first followed by thread one
- Dynamic Partial Order Reduction tries to answer the following question: Are any of these sequences redundant?
Dynamic Partial Order Reduction

- Tests one execution of each equivalence class
- Two sequences are equivalent if they can be obtained by permuting adjacent independent events

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

Possible Executions:
1. `x = 1; y = 1; y = 2`
2. `x = 1; y = 2; y = 1`

- At the center of dynamic partial order reduction is the notion of dependence between events in the program.
- Based on the definition of dependence, different sequences of events can be grouped into equivalence classes.
- After grouping the events into equivalence classes, DPOR simply tests one representative sequence from each equivalence class.
- Consider the following program. There are three possible thread schedules:
  - First: thread one executes until completion followed by thread two
  - Second: Thread one executes one statement, followed by thread two, and then thread one executes
  - And, finally: thread two executes first followed by thread one
- Dynamic Partial Order Reduction tries to answer the following question: Are any of these sequences redundant?
Dynamic Partial Order Reduction

- Tests one execution of each equivalence class
- Two sequences are equivalent if they can be obtained by permuting adjacent independent events

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

Possible Executions:

1. `x = 1; y = 1; y = 2`
2. `x = 1; y = 2; y = 1`
3. `y = 2; x = 1; y = 1`

- At the center of dynamic partial order reduction is the notion of dependence between events in the program.
- Based on the definition of dependence, different sequences of events can be grouped into equivalence classes.
- After grouping the events into equivalence classes, DPOR simply tests one representative sequence from each equivalence class.
- Consider the following program. There are three possible thread schedules:
  - First: thread one executes until completion followed by thread two
  - Second: Thread one executes one statement, followed by thread two, and then thread one executes
  - And, finally: thread two executes first followed by thread one
- Dynamic Partial Order Reduction tries to answer the following question: Are any of these sequences redundant?
Conflict Dependence: Statements are dependent if:

1. Accessing the same memory location,
2. different threads, and
3. at least one is a write.

1. \( x = 1; \ y = 1; \ y = 2 \)
2. \( x = 1; \ y = 2; \ y = 1 \)
3. \( y = 2; \ x = 1; \ y = 1 \)

Final States:

1. \( x = 1, \ y = 2 \)
2. \( x = 1, \ y = 1 \)
3. \( x = 1, \ y = 1 \)

- The simplest, and often used, definition of dependence is conflict dependence.
- Conflict dependence considers statements to be dependent if they are accessing the same memory location, from two different threads, and at least one is a write.
- If we examine the same set of executions using conflict dependence, we can see that two are in the same equivalence class.
- The first two events in sequences two and three are independent as defined by conflict dependence.
- They are from two different threads, accessing different locations in memory.
- We can see this reflected in the final states: for the last two sequences the final state of the program is the same.
- Partial order methods make use of this concept: out of the last two sequences, only one needs to be tested.
Conflict Dependence: Statements are dependent if:

1. Accessing the same memory location,
2. different threads, and
3. at least one is a write.

1. \( x = 1; y = 1; y = 2 \)
2. \( x = 1; y = 2; y = 1 \)
3. \( y = 2; x = 1; y = 1 \)

Final States:

1. \( x = 1, y = 2 \)
2. \( x = 1, y = 1 \)
3. \( x = 1, y = 1 \)

The simplest, and often used, definition of dependence is conflict dependence.

Conflict dependence considers statements to be dependent if they are accessing the same memory location, from two different threads, and at least one is a write.

If we examine the same set of executions using conflict dependence we can see that two are in the same equivalence class.

The first two events in sequences two and three are independent as defined by conflict dependence.

They are from two different threads, accessing different locations in memory.

We can see this reflected in the final states: for the last two sequences the final state of the program is the same.

Partial order methods make use of this concept: out of the last two sequences, only one needs to be tested.
Conflict Dependence: Statements are dependent if:

1. Accessing the same memory location,
2. different threads, and
3. at least one is a write.

1. \(x = 1; y = 1; y = 2\)
2. \(x = 1; y = 2; y = 1\)
3. \(y = 2; x = 1; y = 1\)

Final States:

1. \(x = 1, y = 2\)
2. \(x = 1, y = 1\)
3. \(x = 1, y = 1\)

- The simplest, and often used, definition of dependence is conflict dependence.
- Conflict dependence considers statements to be dependent if they are accessing the same memory location, from two different threads, and at least one is a write.
- If we examine the same set of executions using conflict dependence we can see that two are in the same equivalence class.
- The first two events in sequences two and three are independent as defined by conflict dependence.
- They are from two different threads, accessing different locations in memory.
- We can see this reflected in the final states: for the last two sequences the final state of the program is the same.
- Partial order methods make use of this concept: out of the last two sequences, only one needs to be tested.
Background: Where does conflict dependence fail?

```c
int x, y;
void thread1(void) {
  x = 10;
  // ...
}
void thread2(void) {
  x = 10;
  // ...
}
```

▶ View Dependence: Only reorder transitions if they change the program state

But, conflict dependence is not perfect.
Researchers improved on the idea by defining view dependence.
View dependence considers two statements to be dependent if their order can change program state.
Consider the following program: both threads set the shared variable x to 10.
These two statements are conflict dependent since they are from two different threads, accessing the same memory location, and they are both writes.
However, regardless of their order, the state of the program remains the same.
As a result, using view dependence results in fewer thread schedules required to be tested.
This is the starting point for our contribution. We wondered if we could come up with an even more precise definition of dependence.
A more precise the dependence definition results in fewer transitions to test, resulting in faster runtime.
Background: Where does conflict dependence fail?

```c
int x, y;
void thread1(void) {
    x = 10;
    // ...
}
void thread2(void) {
    x = 10;
    // ...
}
```

- **View Dependence**: Only reorder transitions if they change the program state

- But, conflict dependence is not perfect.
- Researchers improved on the idea by defining view dependence.
- View dependence considers two statements to be dependent if their order can change program state.
- Consider the following program: both threads set the shared variable \( x \) to 10.
- These two statements are conflict dependent since they are from two different threads, accessing the same memory location, and they are both writes.
- However, regardless of their order, the state of the program remains the same.
- As a result, using view dependence results in fewer thread schedules required to be tested.
- This is the starting point for our contribution. We wondered if we could come up with an even more precise definition of dependence.
- A more precise the dependence definition results in fewer transitions to test, resulting in faster runtime.

**View Dependence**

- Only reorder transitions if they change the program state.
Background: Where does conflict dependence fail?

```c
int x, y;
void thread1(void) {
    x = 10;
    // ...
}
void thread2(void) {
    x = 10;
    // ...
}
```

▶ **View Dependence**: Only reorder transitions if they change the program state

Can we do better?

- But, conflict dependence is not perfect.
- Researchers improved on the idea by defining view dependence.
- View dependence considers two statements to be dependent if their order can change program state.
- Consider the following program: both threads set the shared variable `x` to 10.
- These two statements are conflict dependent since they are from two different threads, accessing the same memory location, and they are both writes.
- However, regardless of their order, the state of the program remains the same.
- As a result, using view dependence results in fewer thread schedules required to be tested.
- This is the starting point for our contribution. We wondered if we could come up with an even more precise definition of dependence.
- A more precise the dependence definition results in fewer transitions to test, resulting in faster runtime.
Overview

Motivation

Background
- Static vs. Dynamic Concurrent Program Analysis
- Dynamic Partial Order Reduction

Motivating Example

Our New Method: Predicate Dependence

Experiments

Next, I will go over a motivating example for our work
To understand the intuition behind our method, consider the following program from the International Software Verification Competition.

- Multiple threads are concurrently accessing a shared hash table.
- We would like to verify that none of the threads ever access an index of the hash table which is out-of-bounds.
- First, notice that the assertion will fail based off the value of $h$.
- The value of $h$ depends on the value of $w$.
- The value of $w$ depends on the value of $m$ and $tid$.
- $m$ is a thread local variable.
- $tid$ comes from the arguments passed to the thread.
- And the argument is passed to the thread by `main`.
- Since `main` sets the arguments before the thread is created, there can be no interference between threads amongst all of these variables.
- As a result, for the purpose of checking this assertion, we only need to test one thread schedule rather than thousands.
int NUM_THREADS = 12
int SIZE = 128
int MAX = 4
int table[SIZE];
int thread_routine(int *arg) {
    int tid = ((int*)arg);
    int m = 0, w, h;
    while(1) {
        if (m < MAX) {
            w = (++m) * 11 + tid;
        } else {
            thread_exit(0);
        }
        h = (w * 7) % SIZE;
        if (h < 0) {
            assert(0);
        }
        while (cas(table, h, 0, w) == 0) {
            h = (h+1) % SIZE;
        }
    }
}
int main() {
    for (int i = 0; i < NUM_THREADS; ++i)
        thread_create(thread_routine(i));
    ...
}
Motivating Example

To understand the intuition behind our method, consider the following program from the International Software Verification Competition.

Multiple threads are concurrently accessing a shared hash table.

We would like to verify that none of the threads ever access an index of the hash table which is out-of-bounds.

First, notice that the assertion will fail based off the value of h.

The value of h depends on the value of w.

The value of w depends on the value of m and tid.

m is a thread local variable.

tid comes from the arguments passed to the thread.

And the argument is passed to the thread by main.

Since main sets the arguments before the thread is created, there can be no interference between threads amongst all of these variables.

As a result, for the purpose of checking this assertion, we only need to test one thread schedule rather than thousands.
To understand the intuition behind our method, consider the following program from the International Software Verification Competition.

- Multiple threads are concurrently accessing a shared hash table.
- We would like to verify that none of the threads ever access an index of the hash table which is out-of-bounds.
- First, notice that the assertion will fail based off the value of \( h \).
- The value of \( h \) depends on the value of \( w \).
- The value of \( w \) depends on the value of \( m \) and \( \text{tid} \).
- \( m \) is a thread local variable.
- \( \text{tid} \) comes from the arguments passed to the thread.
- And the argument is passed to the thread by \text{main}.
- Since \text{main} sets the arguments before the thread is created, there can be no interference between threads amongst all of these variables.
- As a result, for the purpose of checking this assertion, we only need to test one thread schedule rather than thousands.
int NUM_THREADS = 12
int SIZE = 128
int MAX = 4
int table[SIZE];
int *thread_routine(int *arg) {
    int tid = ((int*) arg);
    int m = 0, w, h;
    while(1) {
        if (m < MAX)
            w = (++ m) * 11 + tid;
        else
            thread_exit(0);
        h = (w * 7) % SIZE;
        if (h < 0) {
            assert(0);
        }
        while (cas(table, h, 0, w) == 0) {
            h = (h+1) % SIZE;
        }
    }
}
int main () {
    for (int i = 0; i < NUM_THREADS; ++i)
        thread_create(thread_routine(i));
    ...
To understand the intuition behind our method, consider the following program from the International Software Verification Competition.

Multiple threads are concurrently accessing a shared hash table.

We would like to verify that none of the threads ever access an index of the hash table which is out-of-bounds.

First, notice that the assertion will fail based off the value of `h`.

The value of `h` depends on the value of `w`.

The value of `w` depends on the value of `m` and `tid`.

`m` is a thread local variable.

`tid` comes from the arguments passed to the thread.

And the argument is passed to the thread by `main`.

Since `main` sets the arguments before the thread is created, there can be no interference between threads amongst all of these variables.

As a result, for the purpose of checking this assertion, we only need to test one thread schedule rather than thousands.
To understand the intuition behind our method, consider the following program from the International Software Verification Competition.

Multiple threads are concurrently accessing a shared hash table.

We would like to verify that none of the threads ever access an index of the hash table which is out-of-bounds.

First, notice that the assertion will fail based off the value of h.

The value of h depends on the value of w.

The value of w depends on the value of m and tid.

m is a thread local variable.

tid comes from the arguments passed to the thread.

And the argument is passed to the thread by main.

Since main sets the arguments before the thread is created, there can be no interference between threads amongst all of these variables.

As a result, for the purpose of checking this assertion, we only need to test one thread schedule rather than thousands.
Motivating Example

To understand the intuition behind our method, consider the following program from the International Software Verification Competition.

Multiple threads are concurrently accessing a shared hash table.

We would like to verify that none of the threads ever access an index of the hash table which is out-of-bounds.

First, notice that the assertion will fail based off the value of $h$.

The value of $h$ depends on the value of $w$.

The value of $w$ depends on the value of $m$ and $tid$.

$m$ is a thread local variable.

$tid$ comes from the arguments passed to the thread.

And the argument is passed to the thread by main.

Since main sets the arguments before the thread is created, there can be no interference between threads amongst all of these variables.

As a result, for the purpose of checking this assertion, we only need to test one thread schedule rather than thousands.
int NUM_THREADS = 12
int SIZE = 128
int MAX = 4
int table[SIZE];
int *thread_routine(int *arg) {
    int tid = ((int*) arg);
    int m = 0, w, h;
    while(1) {
        if (m < MAX) {
            w = (++m) * 11 + tid;
        } else {
            thread_exit(0);
        }
        h = (w * 7) % SIZE;
        if (h < 0) {
            assert(0);
        }
        while (cas(table, h, 0, w) == 0) {
            h = (h+1) % SIZE;
        }
    }
}
int main() {
    for (int i = 0; i < NUM_THREADS; ++i)
        thread_create(thread_routine(i));
    ...
}
The following graph shows the runtime results for our new method compared to traditional dynamic partial order reduction.

- As the number of threads grows, the runtime of DPOR scales exponentially.
- After 15 threads are in the system, the runtime of DPOR exceeds two hours even for this simple program.
- Our new method, in comparison, is several orders of magnitude faster.
• Next, I will present our new definition of dependence.
• Consider the following program.

• By all previous definitions of dependence, the following transitions are dependent.

• There are two thread schedules in this program: thread one going first, and thread two going first.

• Notice that, regardless of the order of thread one and thread two, the assertion will never be violated since the value of x is always greater than 5.

• As a result, the order of the two writes to x is actually immaterial to checking the validity of the assertion.

• This is the crux of our contribution: we only reorder transitions which could affect the validity of an assertion.
void thread_1(void) {
    x = 20;
    if (x < 5) {
        assert(0);
    }
}

void thread_2(void) {
    x = 10;
}

- Consider the following program.
- By all previous definitions of dependence, the following transitions are dependent.
- There are two thread schedules in this program: thread one going first, and thread two going first.
- Notice that, regardless of the order of thread one and thread two, the assertion will never be violated since the value of x is always greater than 5.
- As a result, the order of the two writes to x is actually immaterial to checking the validity of the assertion.
- This is the crux of our contribution: we only reorder transitions which could affect the validity of an assertion.
Consider the following program.

By all previous definitions of dependence, the following transitions are dependent.

There are two thread schedules in this program: thread one going first, and thread two going first.

Notice that, regardless of the order of thread one and thread two, the assertion will never be violated since the value of x is always greater than 5.

As a result, the order of the two writes to x is actually immaterial to checking the validity of the assertion.

This is the crux of our contribution: we only reorder transitions which could affect the validity of an assertion.
Predicate Dependence

Consider the following program.

By all previous definitions of dependence, the following transitions are dependent.

There are two thread schedules in this program: thread one going first, and thread two going first.

Notice that, regardless of the order of thread one and thread two, the assertion will never be violated since the value of x is always greater than 5.

As a result, the order of the two writes to x is actually immaterial to checking the validity of the assertion.

This is the crux of our contribution: we only reorder transitions which could affect the validity of an assertion.

Our New Method: Predicate Dependence

Predicate Dependence

void thread_1(void) {
  x = 20;
  if (x < 5) {
    assert(0);
  }
}

void thread_2(void) {
  x = 10;
}

1. x = 20; x = 10; if (x < 5)
2. x = 10; x = 20; if (x < 5)
Consider the following program.

By all previous definitions of dependence, the following transitions are dependent.

There are two thread schedules in this program: thread one going first, and thread two going first.

Notice that, regardless of the order of thread one and thread two, the assertion will never be violated since the value of \( x \) is always greater than 5.

As a result, the order of the two writes to \( x \) is actually immaterial to checking the validity of the assertion.

This is the crux of our contribution: we only reorder transitions which could affect the validity of an assertion.
The following is a high level overview of our method.

First, we take a multithreaded C/C++ program and compile it using the LLVM frontend clang.

Next we instrument the code for dynamic analysis.

After that, we statically analyze the program to calculate the dependencies between statements.

Finally, we use the dependency information and the program input to dynamically analyze the program.
Program Slicing

sum = 0

i = 1

sum = add(out)

assert(0)

Our New Method: Predicate Dependence

• We use slicing to calculate the dependencies of each assertion in the program.
• Program slicing works by considering both the control and data dependencies of a statement in the program.
• Consider the following example where we slice on the assert statement
• The only control dependency of the assertion is the check if sum is equal to one.
• The first data dependency is on the return of the add function
• The add function depends on its two inputs.
• In this case, both inputs come from the variables sum and i in main.
• Once we have calculated the control and data dependencies of a statement, as shown in the graph on the left, the slice of a statement is simply a backwards traversal on the statement.
• In this case, the slice contains the entire program
Program Slicing

- We use slicing to calculate the dependencies of each assertion in the program.
- Program slicing works by considering both the control and data dependencies of a statement in the program.
- Consider the following example where we slice on the assert statement:
  - The only control dependency of the assertion is the check if sum is equal to one.
  - The first data dependency is on the return of the add function.
  - The add function depends on its two inputs.
  - In this case, both inputs come from the variables sum and i in main.
  - Once we have calculated the control and data dependencies of a statement, as shown in the graph on the left, the slice of a statement is simply a backwards traversal on the statement.
  - In this case, the slice contains the entire program.
Program Slicing

We use slicing to calculate the dependencies of each assertion in the program.

Program slicing works by considering both the control and data dependencies of a statement in the program.

Consider the following example where we slice on the assert statement:

- The only control dependency of the assertion is the check if sum is equal to one.
- The first data dependency is on the return of the add function.
- The add function depends on its two inputs.
- In this case, both inputs come from the variables sum and i in main.
- Once we have calculated the control and data dependencies of a statement, as shown in the graph on the left, the slice of a statement is simply a backwards traversal on the statement.
- In this case, the slice contains the entire program.
We use slicing to calculate the dependencies of each assertion in the program.

Program slicing works by considering both the control and data dependencies of a statement in the program.

Consider the following example where we slice on the assert statement.

The only control dependency of the assertion is the check if sum is equal to one.

The first data dependency is on the return of the add function.

The add function depends on its two inputs.

In this case, both inputs come from the variables sum and i in main.

Once we have calculated the control and data dependencies of a statement, as shown in the graph on the left, the slice of a statement is simply a backwards traversal on the statement.

In this case, the slice contains the entire program.
Program Slicing

We use slicing to calculate the dependencies of each assertion in the program.

Program slicing works by considering both the control and data dependencies of a statement in the program.

Consider the following example where we slice on the assert statement.

The only control dependency of the assertion is the check if sum is equal to one.

The first data dependency is on the return of the add function.

The add function depends on its two inputs.

In this case, both inputs come from the variables sum and i in main.

Once we have calculated the control and data dependencies of a statement, as shown in the graph on the left, the slice of a statement is simply a backwards traversal on the statement.

In this case, the slice contains the entire program.
Program Slicing

- We use slicing to calculate the dependencies of each assertion in the program.
- Program slicing works by considering both the control and data dependencies of a statement in the program.
- Consider the following example where we slice on the assert statement.
  - The only control dependency of the assertion is the check if sum is equal to one.
  - The first data dependency is on the return of the add function.
  - The add function depends on its two inputs.
  - In this case, both inputs come from the variables sum and i in main.
  - Once we have calculated the control and data dependencies of a statement, as shown in the graph on the left, the slice of a statement is simply a backwards traversal on the statement.
  - In this case, the slice contains the entire program.
Aliasing

Aliasing between program statements adds another level of complexity

```c
int array[12];
void thread1() {
    for (int i = 0; i < 6; ++i) {
        array[i] = array[i] + 1;
    }
}
int main(int argc, char *argv[]) {
    int idx = atoi(argv[1]) % 12;
    thread_create(thread1);
    array[idx] = array[idx] + 1;
}
```

- For simple programs, the previous static slicing method works very well.
- However, for real world programs aliasing between statements creates another complexity.
- Consider the following program where main reads the user’s input to determine the index of a shared array.
- Main then creates a thread which accesses the first 6 elements in the array.
- Finally, main modifies the array at the user specified index.
- Depending on the user’s input, the statements on lines 4 and 10 may or may not access the same location in memory.
- As a result, a conservative static analysis will have to assume that the statements always access the same location in memory.
- Our new method statically calculates the slice while ignoring aliasing and then at runtime calculates the alias information.
Aliasing between program statements adds another level of complexity. Consider the following program where main reads the user’s input to determine the index of a shared array. Main then creates a thread which accesses the first 6 elements in the array. Finally, main modifies the array at the user specified index.

- For simple programs, the previous static slicing method works very well.
- However, for real world programs aliasing between statements creates another complexity.
- Consider the following program where main reads the user’s input to determine the index of a shared array.
- Main then creates a thread which accesses the first 6 elements in the array.
- Finally, main modifies the array at the user specified index.
- Depending on the user’s input, the statements on lines 4 and 10 may or may not access the same location in memory.
- As a result, a conservative static analysis will have to assume that the statements always access the same location in memory.
- Our new method statically calculates the slice while ignoring aliasing and then at runtime calculates the alias information.
Aliasing between program statements adds another level of complexity.

```c
int array[12];
void thread1() {
    for (int i = 0; i < 6; ++i) {
        array[i] = array[i] + 1;
    }
}

int main(int argc, char *argv[]) {
    int idx = atoi(argv[1]) % 12;
    thread_create(thread1);
    array[idx] = array[idx] + 1;
}
```

- Calculate slice while **ignoring** aliasing
- Calculate aliasing information at **runtime**

- For simple programs, the previous static slicing method works very well.
- However, for real world programs aliasing between statements creates another complexity.
- Consider the following program where main reads the user’s input to determine the index of a shared array.
- Main then creates a thread which accesses the first 6 elements in the array.
- Finally, main modifies the array at the user specified index.
- Depending on the user’s input, the statements on lines 4 and 10 may or may not access the same location in memory.
- As a result, a conservative static analysis will have to assume that the statements always access the same location in memory.
- Our new method statically calculates the slice while ignoring aliasing and then at runtime calculates the alias information.
Dealing with Aliasing

- In order to deal with aliasing, we supplement the static analysis at runtime.
- Recall that our method first statically analyzes the program before the dynamic analysis begins.
- We start with a slice which was created while ignoring aliasing.
Dealing with Aliasing

Given:
1. A slice created ignoring aliasing
2. A sequence of transitions executed by the program

Algorithm:

- For each pair of runtime events ($t_i, t_j$)
- If $t_i$ and $t_j$ are accessing the same memory location, and
- $t_i$ is on the slice but $t_j$ is not,
- then, add $t_j$ to the slice.

- Next, we use each sequence of transitions executed by DPOR to expand the slice.
- Specifically, we check each pair of transitions executed at runtime
- If the two transitions are accessing the same memory location and only one of them is on the slice then we add the other transition to the slice.
- The reason this works is that if the two transitions are accessing the same memory location they are aliasing.
- As a result, we avoid the issue of over-approximating the alias analysis statically.
Dealing with Aliasing

Given:
1. A slice created ignoring aliasing
2. A sequence of transitions executed by the program

Algorithm:
- For each pair of runtime events \((t_i, t_j)\)
  - If \(t_i\) and \(t_j\) are accessing the same memory location, and
  - \(t_i\) is on the slice but \(t_j\) is not,
  - then, add \(t_j\) to the slice

Next, we use each sequence of transitions executed by DPOR to expand the slice.

Specifically, we check each pair of transitions executed at runtime.

If the two transitions are accessing the same memory location and only one of them is on the slice then we add the other transition to the slice.

The reason this works is that if the two transitions are accessing the same memory location they are aliasing.

As a result, we avoid the issue of over-approximating the alias analysis statically.
Dealing with Aliasing

Given:
1. A slice created ignoring aliasing
2. A sequence of transitions executed by the program

Algorithm:
- For each pair of runtime events \((t_i, t_j)\)
- If \(t_i\) and \(t_j\) are accessing the same memory location, and
- \(t_i\) is on the slice but \(t_j\) is not,
  - Next, we use each sequence of transitions executed by DPOR to expand the slice.
  - Specifically, we check each pair of transitions executed at runtime
  - If the two transitions are accessing the same memory location and only one of them is on the slice then we add the other transition to the slice.
  - The reason this works is that if the two transitions are accessing the same memory location they are aliasing.
  - As a result, we avoid the issue of over-approximating the alias analysis statically.
Dealing with Aliasing

Given:
1. A slice created ignoring aliasing
2. A sequence of transitions executed by the program

Algorithm:
- For each pair of runtime events \((t_i, t_j)\)
- If \(t_i\) and \(t_j\) are accessing the same memory location, and
- \(t_i\) is on the slice but \(t_j\) is not,
- then, add \(t_j\) to the slice

Next, we use each sequence of transitions executed by DPOR to expand the slice.

Specifically, we check each pair of transitions executed at runtime.

If the two transitions are accessing the same memory location and only one of them is on the slice then we add the other transition to the slice.

The reason this works is that if the two transitions are accessing the same memory location they are aliasing.

As a result, we avoid the issue of over-approximating the alias analysis statically.
int a;
void thread1(void) {
    a = 0; // 3
}
void thread2(void) {
    a = 1; // 6
    if (a != 1) // 7
        assert(0); // 8
}

- As an example of the algorithm, consider the following program:
- Two threads are modifying the shared variable a.
- One of them checks that the value of the variable is equal to 1.
- First, we generate the slice of the assertion while ignoring aliasing and get the following.
- Notice that the crucial write to a by thread 1 is missing from the slice.
- Then, at runtime we expand the slice by examining each sequence of transitions.
- For example, consider that the first sequence run is the following.
- Our algorithm examines each pair from the sequence.
- Consider the first pair selected to be (3, 6). Notice that both statement 3 and statement 6 are accessing the same location in memory.
- Additionally, 6 is on the slice and 3 is not on the slice.
- So we add 3 to the slice.
- After this, our algorithm continues to examine the remaining pairs, but the slice is complete.
Example

```c
int a;
void thread1(void) {
a = 0;    // 3
}
void thread2(void) {
a = 1;    // 6
if (a != 1)  // 7
    assert(0);  // 8
}
```

- As an example of the algorithm, consider the following program:
- Two threads are modifying the shared variable a.
- One of them checks that the value of the variable is equal to 1.
- First, we generate the slice of the assertion while ignoring aliasing and get the following.
- Notice that the crucial write to a by thread 1 is missing from the slice.
- Then, at runtime we expand the slice by examining each sequence of transitions.
- For example, consider that the first sequence run is the following.
- Our algorithm examines each pair from the sequence.
- Consider the first pair selected to be (3,6). Notice that both statement 3 and statement 6 are accessing the same location in memory.
- Additionally, 6 is on the slice and 3 is not on the slice.
- So we add 3 to the slice.
- After this, our algorithm continues to examine the remaining pairs, but the slice is complete.
As an example of the algorithm, consider the following program:

- Two threads are modifying the shared variable `a`.
- One of them checks that the value of the variable is equal to 1.
- First, we generate the slice of the assertion while ignoring aliasing and get the following.
- Notice that the crucial write to `a` by thread 1 is missing from the slice.
- Then, at runtime we expand the slice by examining each sequence of transitions.
- For example, consider that the first sequence run is the following.
- Our algorithm examines each pair from the sequence.
- Consider the first pair selected to be (3,6). Notice that both statement 3 and statement 6 are accessing the same location in memory.
- Additionally, 6 is on the slice and 3 is not on the slice.
- So we add 3 to the slice.
- After this, our algorithm continues to examine the remaining pairs, but the slice is complete.
As an example of the algorithm, consider the following program:

- Two threads are modifying the shared variable `a`.
- One of them checks that the value of the variable is equal to 1.
- First, we generate the slice of the assertion while ignoring aliasing and get the following.
- Notice that the crucial write to `a` by thread 1 is missing from the slice.
- Then, at runtime we expand the slice by examining each sequence of transitions.
- For example, consider that the first sequence run is the following.
- Our algorithm examines each pair from the sequence.
- Consider the first pair selected to be (3,6). Notice that both statement 3 and statement 6 are accessing the same location in memory.
- Additionally, 6 is on the slice and 3 is not on the slice.
- So we add 3 to the slice.
- After this, our algorithm continues to examine the remaining pairs, but the slice is complete.
As an example of the algorithm, consider the following program:

- Two threads are modifying the shared variable a.
- One of them checks that the value of the variable is equal to 1.
- First, we generate the slice of the assertion while ignoring aliasing and get the following.
- Notice that the crucial write to a by thread 1 is missing from the slice.
- Then, at runtime we expand the slice by examining each sequence of transitions.
- For example, consider that the first sequence run is the following.
- Our algorithm examines each pair from the sequence.
- Consider the first pair selected to be (3,6). Notice that both statement 3 and statement 6 are accessing the same location in memory.
- Additionally, 6 is on the slice and 3 is not on the slice.
- So we add 3 to the slice.
- After this, our algorithm continues to examine the remaining pairs, but the slice is complete.

### Example

```c
int a;
void thread1(void) {
    a = 0; // 3
}
void thread2(void) {
    a = 1; // 6
    if (a != 1) // 7
        assert(0); // 8
}
```

- **Sequence:** $S_1 = 3, 6, 7$
- **Slice:** \{8, 7, 6\}

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Pair</th>
<th>Slice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initially:</td>
<td></td>
<td>{8, 7, 6}</td>
</tr>
<tr>
<td>$S_1$</td>
<td>(3, 6)</td>
<td>{8, 7, 6, 3}</td>
</tr>
<tr>
<td>$S_1$</td>
<td>(3, 7)</td>
<td>{8, 7, 6, 3}</td>
</tr>
<tr>
<td>$S_1$</td>
<td>(6, 7)</td>
<td>{8, 7, 6, 3}</td>
</tr>
</tbody>
</table>
Optimizations

1. Critical section peeking
2. Write–Write pruning

Applicable to DPOR, not just our method

We also present two optimizations which are applicable not just for our method but DPOR in general.

One we call critical section peeking and the other write–write pruning.
Optimizations: Critical Section Peeking

- Reordering mutex lock calls reorders all internal statements
- If the statements aren’t dependent, why bother?

```cpp
mutex array_lock;
int array[16];
void thread_1() {
    for (int i = 0; i < 8; ++i) {
        lock(array_lock);
        array[i] = array[i] + 1;
        unlock(array_lock);
    }
}
void thread_2() {
    for (int i = 8; i < 16; ++i) {
        lock(array_lock);
        array[i] = array[i] + 1;
        unlock(array_lock);
    }
}
```

Our New Method: Predicate Dependence

- Previous DPOR implementation consider lock calls to the same mutex to always be dependent.
- At a highlevel, reordering two lock calls reorders all internal statements.
- The intuition of our method is that we do not need to reorder the locks calls if all the internal statements are independent.
- For example, in this program a single mutex protects an entire array
- Thread one is accessing array items 0 through 7 while thread 2 is accessing items 8 through 15.
- Since within each lock-unlock call the threads never access the same memory location we do not need to consider the two locks dependent.
- Using this optimization for this example, we can reduce the number of runs from 12 thousand to one.
Optimizations: Critical Section Peeking

- Reordering mutex lock calls reorders all internal statements
- If the statements aren’t dependent, why bother?

```c
mutex array_lock;
int array[16];
void thread_1() {
    for (int i = 0; i < 8; ++i) {
        lock(array_lock);
        array[i] = array[i] + 1;
        unlock(array_lock);
    }
}

void thread_2() {
    for (int i = 8; i < 16; ++i) {
        lock(array_lock);
        array[i] = array[i] + 1;
        unlock(array_lock);
    }
}
```

Previous DPOR implementation consider lock calls to the same mutex to always be dependent.

- At a high level, reordering two lock calls reorders all internal statements.
- The intuition of our method is that we do not need to reorder the locks calls if all the internal statements are independent.
- For example, in this program a single mutex protects an entire array
- Thread one is accessing array items 0 through 7 while thread 2 is accessing items 8 through 15.
- Since within each lock-unlock call the threads never access the same memory location we do not need to consider the two locks dependent.
- Using this optimization for this example, we can reduce the number of runs from 12 thousand to one.
Optimizations: Critical Section Peeking

- Reordering mutex lock calls reorders all internal statements
- If the statements aren't dependent, why bother?

```c
mutex array_lock;
int array[16];
void thread_1() {
    for (int i = 0; i < 8; ++i) {
        lock(array_lock);
        array[i] = array[i] + 1;
        unlock(array_lock);
    }
}
void thread_2() {
    for (int i = 8; i < 16; ++i) {
        lock(array_lock);
        array[i] = array[i] + 1;
        unlock(array_lock);
    }
}
```

Previous DPOR implementation consider lock calls to the same mutex to always be dependent.
- At a highlevel, reordering two lock calls reorders all internal statements.
- The intuition of our method is that we do not need to reorder the locks calls if all the internal statements are independent.
- For example, in this program a single mutex protects an entire array.
  - Thread one is accessing array items 0 through 7 while thread 2 is accessing items 8 through 15.
  - Since within each lock-unlock call the threads never access the same memory location we do not need to consider the two locks dependent.
  - Using this optimization for this example, we can reduce the number of runs from 12 thousand to one.
Optimizations: Write–Write Pruning

- A blocked thread can only read the last value written to a shared variable.
- From main’s perspective, a can only be 1 or 6.

```c
int a = 0;
void t1_main(){
a = 7;
a = 6;
}
void t2_main(){
a = 0;
a = 1;
}
int main(int argc, char *argv[]){
    thread_create(t1_main);
    thread_create(t2_main);
    thread_join(t1_main);
    thread_join(t2_main);
    assert(a != 7);
    return 0;
}
```

Our other optimization, Write–Write pruning, is based on the following intuition.

- If a thread is blocked, for example while joining another thread, it is only able to read the last value written to a shared variable.
- Here, main spawns two threads which are both modifying a shared variable.
- Main then joins both threads.
- A traditional DPOR implementation would reorder all possible combinations of the writes between both threads.
- However, from the perspective of main at the time of the assertion call, since each thread has already run to completion, the value of a can only be 6 or 1.
- As a result, we only need to reorder the last write from each thread.
Optimizations: Write–Write Pruning

- A blocked thread can only read the last value written to a shared variable
- From main’s perspective, a can only be 1 or 6

```
int a = 0;
void t1_main(){
a = 7;
a = 6;
}
void t2_main() {
a = 0;
a = 1;
}
int main(int argc, char *argv[]) {
    thread_create(t1_main);
    thread_create(t2_main);
    thread_join(t1_main);
    thread_join(t2_main);
    assert(a != 7);
    return 0;
}
```

Our other optimization, Write–Write pruning, is based on the following intuition.

- If a thread is blocked, for example while joining another thread, it is only able to read the last value written to a shared variable.
- Here, main spawns two threads which are both modifying a shared variable.
- Main then joins both threads.
- A traditional DPOR implementation would reorder all possible combinations of the writes between both threads.
- However, from the perspective of main at the time of the assertion call, since each thread has already run to completion, the value of a can only be 6 or 1.
- As a result, we only need to reorder the last write from each thread.
Optimizations: Write–Write Pruning

- A blocked thread can only read the last value written to a shared variable
- From main’s perspective, a can only be 1 or 6

```c
int a = 0;
void t1_main(){
a = 7;
a = 6;
}
void t2_main() {
a = 0;
a = 1;
}
int main(int argc, char *argv[]) {
thread_create(t1_main);
thread_create(t2_main);
thread_join(t1_main);
thread_join(t2_main);
assert(a != 7);
return 0;
}
```

- Our other optimization, Write–Write pruning, is based on the following intuition.
- If a thread is blocked, for example while joining another thread, it is only able to read the last value written to a shared variable.
- Here, main spawns two threads which are both modifying a shared variable.
- Main then joins both threads.
- A traditional DPOR implementation would reorder all possible combinations of the writes between both threads.
- However, from the perspective of main at the time of the assertion call, since each thread has already run to completion, the value of a can only be 6 or 1.
- As a result, we only need to reorder the last write from each thread.
A blocked thread can only read the last value written to a shared variable.

From main’s perspective, a can only be 1 or 6.

```c
int a = 0;
void t1_main() {
    a = 7;
    a = 6;
}
void t2_main() {
    a = 0;
    a = 1;
}
int main(int argc, char *argv[]) {
    thread_create(t1_main);
    thread_create(t2_main);
    thread_join(t1_main);
    thread_join(t2_main);
    assert(a != 7);
    return 0;
}
```

Our other optimization, Write–Write pruning, is based on the following intuition.

If a thread is blocked, for example while joining another thread, it is only able to read the last value written to a shared variable.

Here, main spawns two threads which are both modifying a shared variable.

Main then joins both threads.

A traditional DPOR implementation would reorder all possible combinations of the writes between both threads.

However, from the perspective of main at the time of the assertion call, since each thread has already run to completion, the value of a can only be 6 or 1.

As a result, we only need to reorder the last write from each thread.
Next I’ll provide a summary of our experimental results.
Experiments

- Tests run on a consumer laptop (Intel i5-3230M)
- C/C++ programs compiled with Clang
- Static analysis done using LLVM
- Limited testing time to two hours
- Tests programs taken from SVCOMP and real world open source programs

Our experiments were performed on a consumer laptop with a mobile i5 processor

We ran our code on C/C++ programs

We used the Clang frontend to LLVM to perform static analysis.

For each test, we allotted two hours runtime

We performed our tests on benchmarks from the International Software Verification Competition and from real world open source programs
• We ran our method on 46 programs to compare it to DPOR.
• For some programs, we did not have any reduction in the number of runs.
• For these cases, our new method does not incur too high of a runtime overhead.
• However, for those programs where we could find a reduction, we often were able to significantly reduce the runtime.
For example, the following are the results for our method running on a nonblocking hashtable.

The results show that our new method can reduce the number of runs required significantly.

After only 7 concurrent operations by each thread, DPOR exceeds the two hour time limit while our method can finish within a reasonable amount of time.
### Runtime Results

For all the tests were we could find a reduction, the following table summarizes the runtime results.

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Time (s)</th>
<th>DPOR</th>
<th>Pred-DPOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccountBad</td>
<td>60</td>
<td>0.05</td>
<td>0.05</td>
<td></td>
</tr>
<tr>
<td>BluetoothBad</td>
<td>88</td>
<td>0.19</td>
<td>0.11</td>
<td></td>
</tr>
<tr>
<td>ReadWrite</td>
<td>50</td>
<td>0.07</td>
<td>0.03</td>
<td></td>
</tr>
<tr>
<td>ReadWriteLock</td>
<td>55</td>
<td>22.03</td>
<td>0.41</td>
<td></td>
</tr>
<tr>
<td>Stateful</td>
<td>54</td>
<td>0.02</td>
<td>0.02</td>
<td></td>
</tr>
<tr>
<td>IndexerSafe12</td>
<td>92</td>
<td>0.57</td>
<td>0.15</td>
<td></td>
</tr>
<tr>
<td>IndexerSafe13</td>
<td>92</td>
<td>4.64</td>
<td>0.17</td>
<td></td>
</tr>
<tr>
<td>IndexerSafe14</td>
<td>92</td>
<td>36.03</td>
<td>0.22</td>
<td></td>
</tr>
<tr>
<td>IndexerSafe15</td>
<td>92</td>
<td>282.06</td>
<td>0.43</td>
<td></td>
</tr>
<tr>
<td>nbds-list</td>
<td>1887</td>
<td>X</td>
<td>0.19</td>
<td></td>
</tr>
<tr>
<td>nbds-hashtable4</td>
<td>2375</td>
<td>37.11</td>
<td>25.02</td>
<td></td>
</tr>
<tr>
<td>nbds-hashtable5</td>
<td>2375</td>
<td>68.05</td>
<td>25.30</td>
<td></td>
</tr>
<tr>
<td>nbds-hashtable6</td>
<td>2375</td>
<td>1082.00</td>
<td>37.74</td>
<td></td>
</tr>
<tr>
<td>nbds-hashtable7</td>
<td>2375</td>
<td>1828.59</td>
<td>73.08</td>
<td></td>
</tr>
<tr>
<td>nbds-hashtable8</td>
<td>2375</td>
<td>X</td>
<td>138.04</td>
<td></td>
</tr>
<tr>
<td>nbds-hashw01</td>
<td>2322</td>
<td>39.36</td>
<td>46.69</td>
<td></td>
</tr>
<tr>
<td>nbds-hashw02</td>
<td>2322</td>
<td>1173.25</td>
<td>1211.83</td>
<td></td>
</tr>
<tr>
<td>nbds-hashw03</td>
<td>2234</td>
<td>942.99</td>
<td>7.66</td>
<td></td>
</tr>
<tr>
<td>nbds-skiplistU1</td>
<td>1942</td>
<td>1.05</td>
<td>3.14</td>
<td></td>
</tr>
<tr>
<td>nbds-skiplistU2</td>
<td>1942</td>
<td>43.67</td>
<td>49.40</td>
<td></td>
</tr>
<tr>
<td>nbds-skiplist</td>
<td>1994</td>
<td>X</td>
<td>0.21</td>
<td></td>
</tr>
<tr>
<td>nedmalloc</td>
<td>6303</td>
<td>X</td>
<td>9.138</td>
<td></td>
</tr>
<tr>
<td>pfscan</td>
<td>934</td>
<td>X</td>
<td>61.24</td>
<td></td>
</tr>
</tbody>
</table>

Especially for larger programs, we were able to achieve a significant reduction in runtime.

For example, many of the programs where DPOR timed out, marked with an X in this table, we were able to finish in under a minute.
Conclusions

- Static–dynamic concurrent program analysis
- Sound verification of assertion violations
- Experimentally, our new method out performs DPOR
- Verifies previously intractable programs

In conclusion, we present a sound optimization for verifying multithreaded programs.

In comparison to DPOR, our method can offer a significant reduction in runtime.

As a result, we were able to verify some programs which were previously intractable.
Cormac Flanagan and Patrice Godefroid.
Dynamic partial-order reduction for model checking software.
In Proceedings of the 32Nd ACM SIGPLAN-SIGACT
Symposium on Principles of Programming Languages, POPL '05, pages 110–121, New York, NY, USA, 2005. ACM.

Azadeh Farzan and Zachary Kincaid.
Verification of parameterized concurrent programs by modular reasoning about data and control.

Patrice Godefroid.