Flow-sensitive Composition of Thread-modular Abstract Interpretation

Markus Kusano Chao Wang

Virginia Tech, ECE Department

January 23, 2017

janatechnology.wordpress.com

- Concurrency bugs have been at the center of many high consequence events
- For example, in the 80s a concurrency bug in a Therac-25 radiation therapy machine lead to at least six accidents
- The bug caused radiation dosages to exceed one thousand times the correct amount
- This year, a concurrency bug was found in the design of firmware protection in Intel processors
- Specifically, the bug enabled an attacker to write to the processors firmware allowing arbitrary malicious code to be executed
- This attack on the processors' firmware is particularly evil since it allows a backdoor to be added to the computer which is persistent across OS re-installations

janatechnology.wordpress.com

softpedia.com

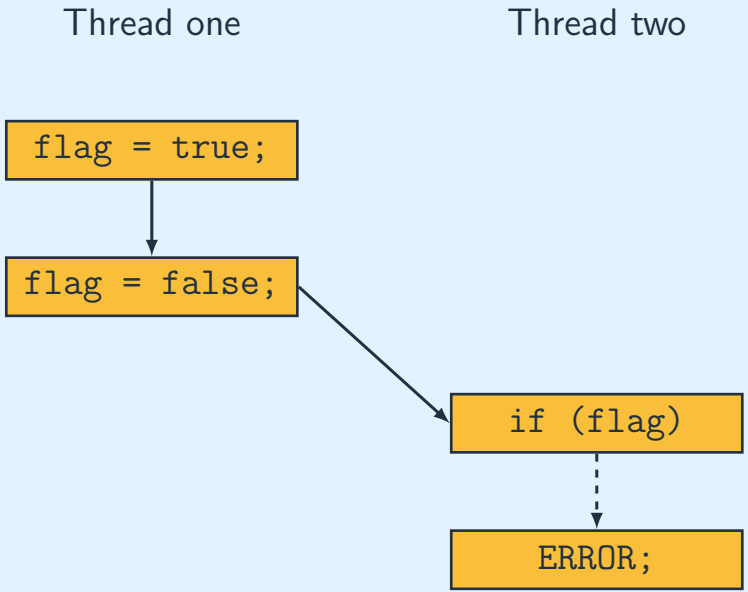
- Concurrency bugs have been at the center of many high consequence events
- For example, in the 80s a concurrency bug in a Therac-25 radiation therapy machine lead to at least six accidents
- The bug caused radiation dosages to exceed one thousand times the correct amount
- This year, a concurrency bug was found in the design of firmware protection in Intel processors
- Specifically, the bug enabled an attacker to write to the processors firmware allowing arbitrary malicious code to be executed
- This attack on the processors' firmware is particularly evil since it allows a backdoor to be added to the computer which is persistent across OS re-installations

Race Conditions

Thread one	Thread two
`flag = true;`	`if (flag)`
`flag = false;`	`ERROR;`

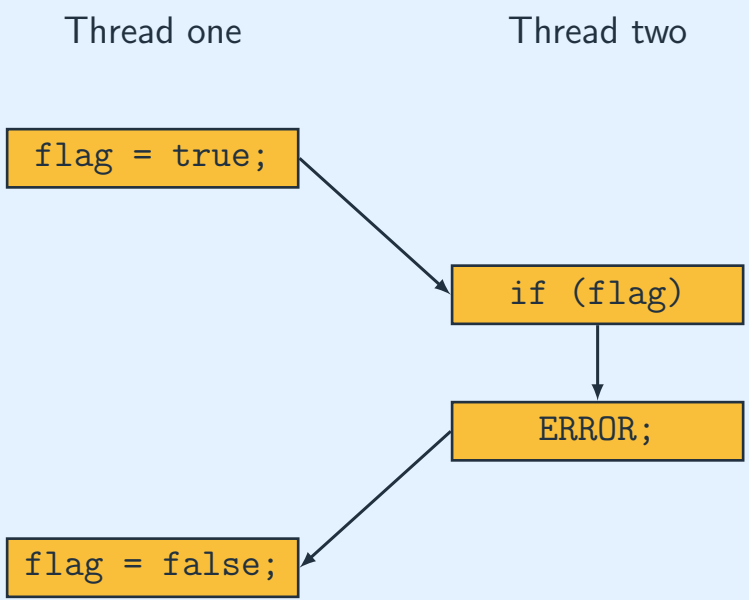
- Both of these previous bugs were caused by race conditions which were undiscovered in the traditional testing
- As an example, suppose we have two threads executing concurrently; one sets the value of flag to true then false, and the other reads the value of flag to determine if it enters a branch
- Additionally, thread two reading the value of true from flag constitutes and error
- A race condition is a bug which is exposed only on certain thread schedules
- For example, if thread one executes before thread two then the error statement is not reachable
- However, if thread one partly executes and then is preempted by thread two, the error is reachable
- So, depending on the non-determinism in the scheduler, the bug is sometimes reachable and sometimes not
- Although this example may seem simple, it is structurally the same as the one causing the Intel bug.

Race Conditions



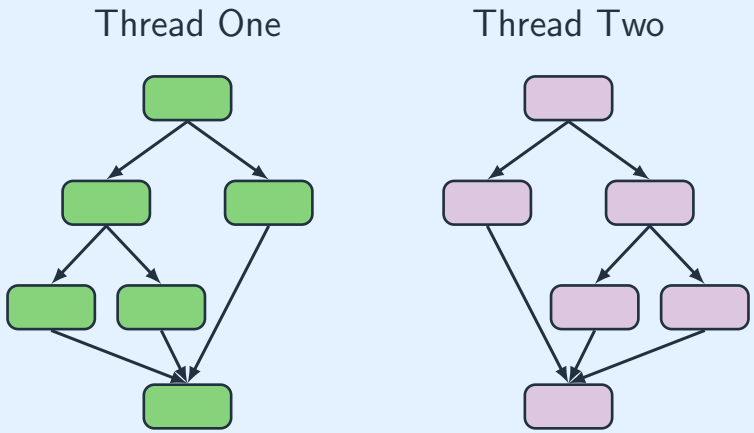
- Both of these previous bugs were caused by race conditions which were undiscovered in the traditional testing
- As an example, suppose we have two threads executing concurrently; one sets the value of flag to true then false, and the other reads the value of flag to determine if it enters a branch
- Additionally, thread two reading the value of true from flag constitutes and error
- A race condition is a bug which is exposed only on certain thread schedules
- For example, if thread one executes before thread two then the error statement is not reachable
- However, if thread one partly executes and then is preempted by thread two, the error is reachable
- So, depending on the non-determinism in the scheduler, the bug is sometimes reachable and sometimes not
- Although this example may seem simple, it is structurally the same as the one causing the Intel bug.

Race Conditions



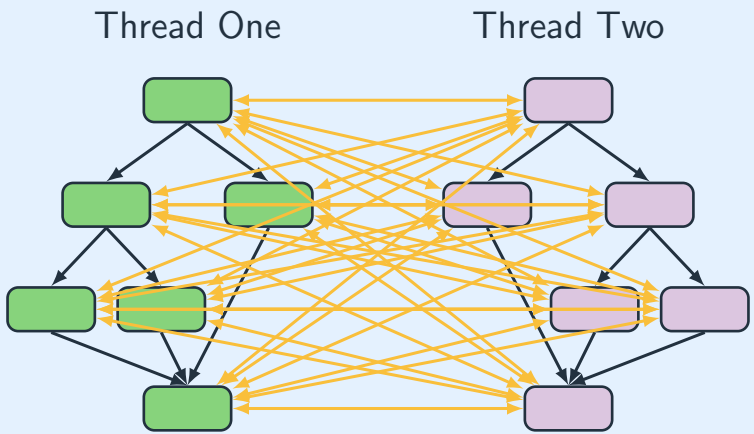
- Both of these previous bugs were caused by race conditions which were undiscovered in the traditional testing
- As an example, suppose we have two threads executing concurrently; one sets the value of flag to true then false, and the other reads the value of flag to determine if it enters a branch
- Additionally, thread two reading the value of true from flag constitutes and error
- A race condition is a bug which is exposed only on certain thread schedules
- For example, if thread one executes before thread two then the error statement is not reachable
- However, if thread one partly executes and then is preempted by thread two, the error is reachable
- So, depending on the non-determinism in the scheduler, the bug is sometimes reachable and sometimes not
- Although this example may seem simple, it is structurally the same as the one causing the Intel bug.

State Explosion



- The reason why concurrency bugs are difficult to detect and reason about is the so called state explosion problem
- Suppose we have the control-flow graphs of two threads on the left and right of this figure
- Each control-flow graph defines all the paths within each thread
- The inter-thread control-flow graph can be generated by taking the composition of the two threads
- In other words, at each point within a thread either the thread itself can execute or the scheduler can change to another thread
- Here, we can see why concurrency bugs are difficult to reason about and detect: even for this tiny example there is a large amount of inter-thread transitions
- Specifically, the number of edges in the graph grows exponentially with respect to the number of threads
- The bug may occur in only on handful of paths through the inter-thread graph, so, because of the graph's size, the probability of detecting a bug using traditional testing approaches is very low

State Explosion



- The reason why concurrency bugs are difficult to detect and reason about is the so called state explosion problem
- Suppose we have the control-flow graphs of two threads on the left and right of this figure
- Each control-flow graph defines all the paths within each thread
- The inter-thread control-flow graph can be generated by taking the composition of the two threads
- In other words, at each point within a thread either the thread itself can execute or the scheduler can change to another thread
- Here, we can see why concurrency bugs are difficult to reason about and detect: even for this tiny example there is a large amount of inter-thread transitions
- Specifically, the number of edges in the graph grows exponentially with respect to the number of threads
- The bug may occur in only on handful of paths through the inter-thread graph, so, because of the graph's size, the probability of detecting a bug using traditional testing approaches is very low

State Explosion Consequences

- Functional testing: poor coverage

- So, what are the consequences of the state explosion problem caused by scheduling non-determinism?
- The traditional functional testing approach is to supply inputs to the program, run it, and then check if it crashes
- Using this approach on a concurrent program, however, has poor coverage: it only explores a small fraction of the possible thread-schedules
- In the sequential world, there are also more heavy-weight analysis techniques such as symbolic execution, abstract interpretation, and model checking
- At a very high level, all of these techniques analyze the semantics of the control-flow graph of the program
- Because of this, the complexity of these approaches grows with respect to the size of the control-flow graph
- As we just saw, since as the number of threads grows the size of the graph grows exponentially, these techniques have difficulty scaling to reasonably sized concurrent programs
- So, the state explosion problem leaves us in a poor situation: our analysis techniques either have poor coverage, or poor scalability
- What is missing is a nice middle ground analysis which is able to have good scalability while providing accurate results

State Explosion Consequences

- Functional testing: poor coverage
- Symblolic execution, abstract interpretation, model checking

- So, what are the consequences of the state explosion problem caused by scheduling non-determinism?
- The traditional functional testing approach is to supply inputs to the program, run it, and then check if it crashes
- Using this approach on a concurrent program, however, has poor coverage: it only explores a small fraction of the possible thread-schedules
- In the sequential world, there are also more heavy-weight analysis techniques such as symbolic execution, abstract interpretation, and model checking
- At a very high level, all of these techniques analyze the semantics of the control-flow graph of the program
- Because of this, the complexity of these approaches grows with respect to the size of the control-flow graph
- As we just saw, since as the number of threads grows the size of the graph grows exponentially, these techniques have difficulty scaling to reasonably sized concurrent programs
- So, the state explosion problem leaves us in a poor situation: our analysis techniques either have poor coverage, or poor scalability
- What is missing is a nice middle ground analysis which is able to have good scalability while providing accurate results

State Explosion Consequences

- Functional testing: poor coverage
- Symblolic execution, abstract interpretation, model checking
  - Analyze control-flow graph semantics

- So, what are the consequences of the state explosion problem caused by scheduling non-determinism?
- The traditional functional testing approach is to supply inputs to the program, run it, and then check if it crashes
- Using this approach on a concurrent program, however, has poor coverage: it only explores a small fraction of the possible thread-schedules
- In the sequential world, there are also more heavy-weight analysis techniques such as symbolic execution, abstract interpretation, and model checking
- At a very high level, all of these techniques analyze the semantics of the control-flow graph of the program
- Because of this, the complexity of these approaches grows with respect to the size of the control-flow graph
- As we just saw, since as the number of threads grows the size of the graph grows exponentially, these techniques have difficulty scaling to reasonably sized concurrent programs
- So, the state explosion problem leaves us in a poor situation: our analysis techniques either have poor coverage, or poor scalability
- What is missing is a nice middle ground analysis which is able to have good scalability while providing accurate results

State Explosion Consequences

- Functional testing: poor coverage
- Symblolic execution, abstract interpretation, model checking
  - Analyze control-flow graph semantics
  - Complexity w.r.t. graph size

- So, what are the consequences of the state explosion problem caused by scheduling non-determinism?
- The traditional functional testing approach is to supply inputs to the program, run it, and then check if it crashes
- Using this approach on a concurrent program, however, has poor coverage: it only explores a small fraction of the possible thread-schedules
- In the sequential world, there are also more heavy-weight analysis techniques such as symbolic execution, abstract interpretation, and model checking
- At a very high level, all of these techniques analyze the semantics of the control-flow graph of the program
- Because of this, the complexity of these approaches grows with respect to the size of the control-flow graph
- As we just saw, since as the number of threads grows the size of the graph grows exponentially, these techniques have difficulty scaling to reasonably sized concurrent programs
- So, the state explosion problem leaves us in a poor situation: our analysis techniques either have poor coverage, or poor scalability
- What is missing is a nice middle ground analysis which is able to have good scalability while providing accurate results

State Explosion Consequences

- Functional testing: poor coverage
- Symblolic execution, abstract interpretation, model checking
  - Analyze control-flow graph semantics
  - Complexity w.r.t. graph size
  - Not scalable

- So, what are the consequences of the state explosion problem caused by scheduling non-determinism?
- The traditional functional testing approach is to supply inputs to the program, run it, and then check if it crashes
- Using this approach on a concurrent program, however, has poor coverage: it only explores a small fraction of the possible thread-schedules
- In the sequential world, there are also more heavy-weight analysis techniques such as symbolic execution, abstract interpretation, and model checking
- At a very high level, all of these techniques analyze the semantics of the control-flow graph of the program
- Because of this, the complexity of these approaches grows with respect to the size of the control-flow graph
- As we just saw, since as the number of threads grows the size of the graph grows exponentially, these techniques have difficulty scaling to reasonably sized concurrent programs
- So, the state explosion problem leaves us in a poor situation: our analysis techniques either have poor coverage, or poor scalability
- What is missing is a nice middle ground analysis which is able to have good scalability while providing accurate results

State Explosion Consequences

- Functional testing: poor coverage
- Symblolic execution, abstract interpretation, model checking
  - Analyze control-flow graph semantics
  - Complexity w.r.t. graph size
  - Not scalable

- So, what are the consequences of the state explosion problem caused by scheduling non-determinism?
- The traditional functional testing approach is to supply inputs to the program, run it, and then check if it crashes
- Using this approach on a concurrent program, however, has poor coverage: it only explores a small fraction of the possible thread-schedules
- In the sequential world, there are also more heavy-weight analysis techniques such as symbolic execution, abstract interpretation, and model checking
- At a very high level, all of these techniques analyze the semantics of the control-flow graph of the program
- Because of this, the complexity of these approaches grows with respect to the size of the control-flow graph
- As we just saw, since as the number of threads grows the size of the graph grows exponentially, these techniques have difficulty scaling to reasonably sized concurrent programs
- So, the state explosion problem leaves us in a poor situation: our analysis techniques either have poor coverage, or poor scalability
- What is missing is a nice middle ground analysis which is able to have good scalability while providing accurate results

State Explosion Consequences

- Functional testing: poor coverage
- Symblolic execution, abstract interpretation, model checking
  - Analyze control-flow graph semantics
  - Complexity w.r.t. graph size
  - Not scalable

- So, what are the consequences of the state explosion problem caused by scheduling non-determinism?
- The traditional functional testing approach is to supply inputs to the program, run it, and then check if it crashes
- Using this approach on a concurrent program, however, has poor coverage: it only explores a small fraction of the possible thread-schedules
- In the sequential world, there are also more heavy-weight analysis techniques such as symbolic execution, abstract interpretation, and model checking
- At a very high level, all of these techniques analyze the semantics of the control-flow graph of the program
- Because of this, the complexity of these approaches grows with respect to the size of the control-flow graph
- As we just saw, since as the number of threads grows the size of the graph grows exponentially, these techniques have difficulty scaling to reasonably sized concurrent programs
- So, the state explosion problem leaves us in a poor situation: our analysis techniques either have poor coverage, or poor scalability
- What is missing is a nice middle ground analysis which is able to have good scalability while providing accurate results

Contributions

- More accurate thread-modular abstract interpretation

- This brings us to our contributions
- We present a new thread-modular abstract interpretation algorithm
- Because the analysis is thread modular, as I will show shortly, we avoid having to compose the threads explicitly and thus avoid the state explosion problem
- Compared to prior work on thread-modular abstract interpretation, our technique is more accurate: in our experiments, we were able to generate 28 times fewer false alarms
- Additionally, this large gain in accuracy came with only a modest increase in runtime overhead

Contributions

- More accurate thread-modular abstract interpretation
- Less false-alarms (28x)

- This brings us to our contributions
- We present a new thread-modular abstract interpretation algorithm
- Because the analysis is thread modular, as I will show shortly, we avoid having to compose the threads explicitly and thus avoid the state explosion problem
- Compared to prior work on thread-modular abstract interpretation, our technique is more accurate: in our experiments, we were able to generate 28 times fewer false alarms
- Additionally, this large gain in accuracy came with only a modest increase in runtime overhead

Contributions

- More accurate thread-modular abstract interpretation
- Less false-alarms (28x)
- Moderate overhead (3x)

- This brings us to our contributions
- We present a new thread-modular abstract interpretation algorithm
- Because the analysis is thread modular, as I will show shortly, we avoid having to compose the threads explicitly and thus avoid the state explosion problem
- Compared to prior work on thread-modular abstract interpretation, our technique is more accurate: in our experiments, we were able to generate 28 times fewer false alarms
- Additionally, this large gain in accuracy came with only a modest increase in runtime overhead

Overview

Next, I'll give some background, starting with what numerical abstract interpretation is, then introducing prior thread-modular analyses

Numerical Abstract Interpretation

```
1 void func(int x, int y) {
2   if (x >= 20) {
3     y = 10;
4   }
5   else {
6     y = 20;
7   }
8   assert(y > 5);
9 }
```


- First, I'll go over numerical abstract interpretation applied to sequential programs
- As hinted by the name, the goal of numerical abstract interpretation is to reason about the numerical properties of a program
- Lets look at how a numerical analysis could be applied to this function
- First, we consider that the user of this function may pass any input values, so, the values of x and y are unconstrained
- Then, we traverse the control-flow graph of the function and update these inputs values for x and y based on the semantics of each statement
- So, if we enter the first branch, we know the value of x is greater than or equal to 20 and the value of y is updated to be 10
- Similarly, in the second branch, the value of x is less than 20 and the value of y is set to 20
- Finally, after exploring both branches, we join the results and get the fact that y is between 10 and 20 and x is unconstrained
- We would like to verify that at the end of the function the value of y is greater than 5
- As we can see from the results, the value of y is larger than 5 so we know that on all possible executions of this function the assertion condition will not be violated
- This technique allows, in general, for automated proofs such as this to be generated

Numerical Abstract Interpr

$x = [-\infty, \infty]$
$y = [-\infty, \infty]$

```
1 void func(int x, int y) {
2   if (x >= 20) {
3     y = 10;
4   }
5   else {
6     y = 20;
7   }
8   assert(y > 5);
9 }
```


- First, I'll go over numerical abstract interpretation applied to sequential programs
- As hinted by the name, the goal of numerical abstract interpretation is to reason about the numerical properties of a program
- Lets look at how a numerical analysis could be applied to this function
- First, we consider that the user of this function may pass any input values, so, the values of x and y are unconstrained
- Then, we traverse the control-flow graph of the function and update these inputs values for x and y based on the semantics of each statement
- So, if we enter the first branch, we know the value of x is greater than or equal to 20 and the value of y is updated to be 10
- Similarly, in the second branch, the value of x is less than 20 and the value of y is set to 20
- Finally, after exploring both branches, we join the results and get the fact that y is between 10 and 20 and x is unconstrained
- We would like to verify that at the end of the function the value of y is greater than 5
- As we can see from the results, the value of y is larger than 5 so we know that on all possible executions of this function the assertion condition will not be violated
- This technique allows, in general, for automated proofs such as this to be generated

Numerical Abstract Interpr

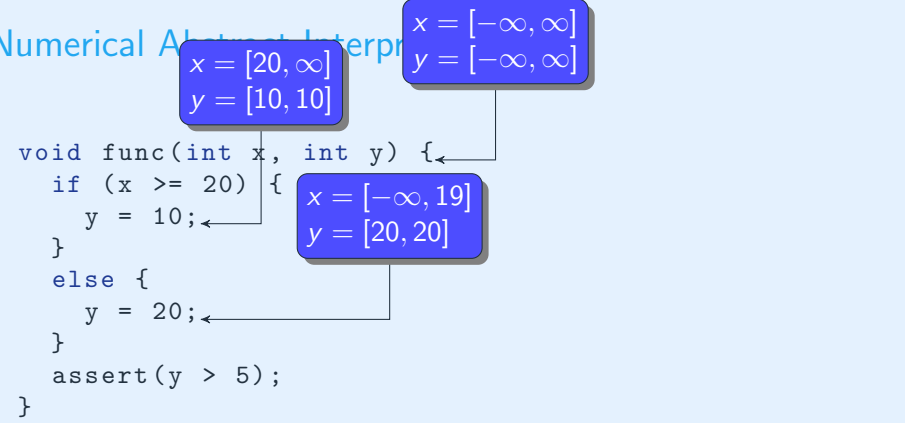
$x = [-\infty, \infty]$
$y = [-\infty, \infty]$

$x = [20, \infty]$
$y = [10, 10]$

```
1 void func(int x, int y) {
2   if (x >= 20) {
3     y = 10;
4   }
5   else {
6     y = 20;
7   }
8   assert(y > 5);
9 }
```

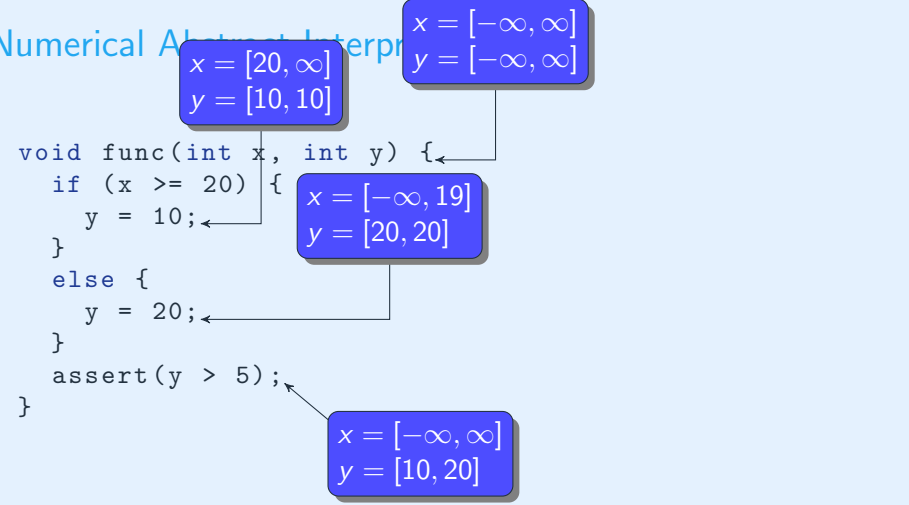

- First, I'll go over numerical abstract interpretation applied to sequential programs
- As hinted by the name, the goal of numerical abstract interpretation is to reason about the numerical properties of a program
- Lets look at how a numerical analysis could be applied to this function
- First, we consider that the user of this function may pass any input values, so, the values of x and y are unconstrained
- Then, we traverse the control-flow graph of the function and update these inputs values for x and y based on the semantics of each statement
- So, if we enter the first branch, we know the value of x is greater than or equal to 20 and the value of y is updated to be 10
- Similarly, in the second branch, the value of x is less than 20 and the value of y is set to 20
- Finally, after exploring both branches, we join the results and get the fact that y is between 10 and 20 and x is unconstrained
- We would like to verify that at the end of the function the value of y is greater than 5
- As we can see from the results, the value of y is larger than 5 so we know that on all possible executions of this function the assertion condition will not be violated
- This technique allows, in general, for automated proofs such as this to be generated

Numerical Abstract Interpr



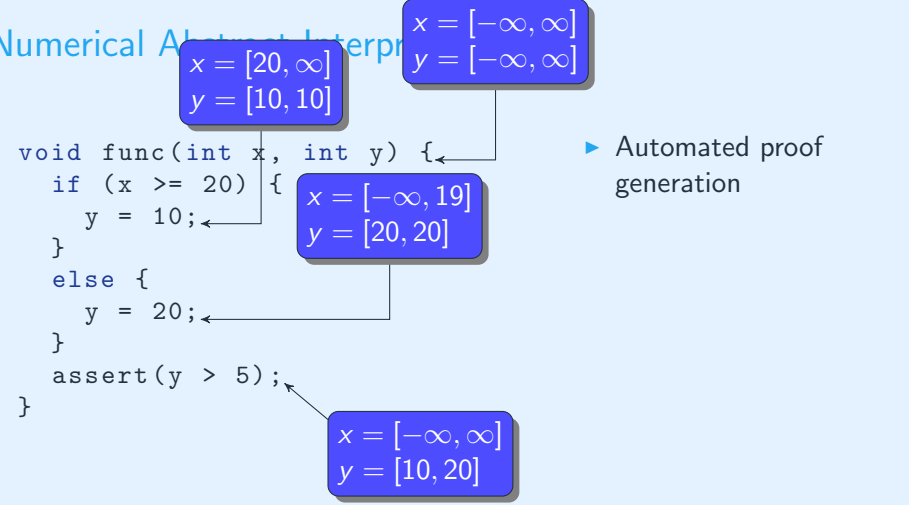
- First, I'll go over numerical abstract interpretation applied to sequential programs
- As hinted by the name, the goal of numerical abstract interpretation is to reason about the numerical properties of a program
- Lets look at how a numerical analysis could be applied to this function
- First, we consider that the user of this function may pass any input values, so, the values of x and y are unconstrained
- Then, we traverse the control-flow graph of the function and update these inputs values for x and y based on the semantics of each statement
- So, if we enter the first branch, we know the value of x is greater than or equal to 20 and the value of y is updated to be 10
- Similarly, in the second branch, the value of x is less than 20 and the value of y is set to 20
- Finally, after exploring both branches, we join the results and get the fact that y is between 10 and 20 and x is unconstrained
- We would like to verify that at the end of the function the value of y is greater than 5
- As we can see from the results, the value of y is larger than 5 so we know that on all possible executions of this function the assertion condition will not be violated
- This technique allows, in general, for automated proofs such as this to be generated

Numerical Abstract Interpr



- First, I'll go over numerical abstract interpretation applied to sequential programs
- As hinted by the name, the goal of numerical abstract interpretation is to reason about the numerical properties of a program
- Lets look at how a numerical analysis could be applied to this function
- First, we consider that the user of this function may pass any input values, so, the values of x and y are unconstrained
- Then, we traverse the control-flow graph of the function and update these inputs values for x and y based on the semantics of each statement
- So, if we enter the first branch, we know the value of x is greater than or equal to 20 and the value of y is updated to be 10
- Similarly, in the second branch, the value of x is less than 20 and the value of y is set to 20
- Finally, after exploring both branches, we join the results and get the fact that y is between 10 and 20 and x is unconstrained
- We would like to verify that at the end of the function the value of y is greater than 5
- As we can see from the results, the value of y is larger than 5 so we know that on all possible executions of this function the assertion condition will not be violated
- This technique allows, in general, for automated proofs such as this to be generated

Numerical Abstract Interpr



- Automated proof generation

- First, I'll go over numerical abstract interpretation applied to sequential programs
- As hinted by the name, the goal of numerical abstract interpretation is to reason about the numerical properties of a program
- Lets look at how a numerical analysis could be applied to this function
- First, we consider that the user of this function may pass any input values, so, the values of x and y are unconstrained
- Then, we traverse the control-flow graph of the function and update these inputs values for x and y based on the semantics of each statement
- So, if we enter the first branch, we know the value of x is greater than or equal to 20 and the value of y is updated to be 10
- Similarly, in the second branch, the value of x is less than 20 and the value of y is set to 20
- Finally, after exploring both branches, we join the results and get the fact that y is between 10 and 20 and x is unconstrained
- We would like to verify that at the end of the function the value of y is greater than 5
- As we can see from the results, the value of y is larger than 5 so we know that on all possible executions of this function the assertion condition will not be violated
- This technique allows, in general, for automated proofs such as this to be generated

Thread Modular Numerical Abstract Interpretation

Initially: `x = 0`

```
11 void thread1() {
12   x = 5;
13 }
```

```
21 void thread2() {
22   int t1 = x;
23   assert(x != 10);
24 }
```

- Analyze threads in isolation

- Next, we'll look at how the previously described sequential analysis can be lifted to a thread-modular analysis
- The gist of the thread-modular numerical analysis is that each thread is analyzed in isolation, then, the results of the thread-local analysis are propagated to all other threads
- Consider the analysis of this two threaded program
- If we first analyze thread one in isolation we can see it has a store on the shared variable x with the value of 5
- Similarly, analyzing thread two, we see a shared variable read of x. Since the thread is analyzed in isolation, thread two can only see the initial value of x which is zero
- Then, after analyzing each thread we propagate the values stored into shared memory across threads
- Specifically, we analyze thread two in the presense of the interfering store from thread one: this means thread two could either read the initial value or the value from thread one
- So, in thread two we determine the value of x to be between 0 and 5 which is sufficient to verify that the property that x is not equal to 10 is never violated

Thread Modular Numerical Abstract Interpretation

Initially: `x = 0`

```
11 void thread1() {
12   x = 5;
13 }
```

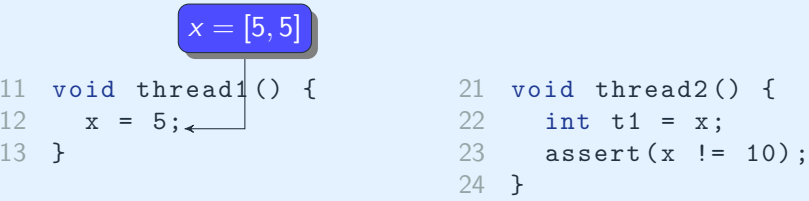
```
21 void thread2() {
22   int t1 = x;
23   assert(x != 10);
24 }
```

- Analyze threads in isolation
- Propagate *interfering* stores to loads

- Next, we'll look at how the previously described sequential analysis can be lifted to a thread-modular analysis
- The gist of the thread-modular numerical analysis is that each thread is analyzed in isolation, then, the results of the thread-local analysis are propagated to all other threads
- Consider the analysis of this two threaded program
- If we first analyze thread one in isolation we can see it has a store on the shared variable x with the value of 5
- Similarly, analyzing thread two, we see a shared variable read of x. Since the thread is analyzed in isolation, thread two can only see the initial value of x which is zero
- Then, after analyzing each thread we propagate the values stored into shared memory across threads
- Specifically, we analyze thread two in the presense of the interfering store from thread one: this means thread two could either read the initial value or the value from thread one
- So, in thread two we determine the value of x to be between 0 and 5 which is sufficient to verify that the property that x is not equal to 10 is never violated

Thread Modular Numerical Abstract Interpretation

Initially: `x = 0`

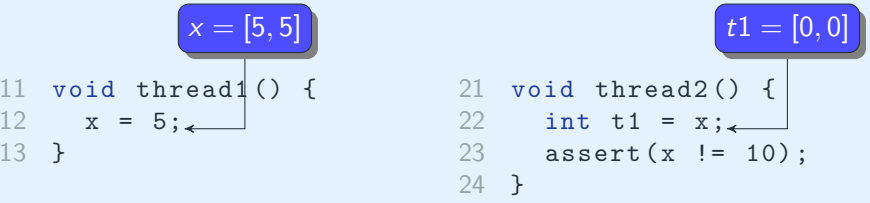


- Analyze threads in isolation
- Propagate *interfering* stores to loads

- Next, we'll look at how the previously described sequential analysis can be lifted to a thread-modular analysis
- The gist of the thread-modular numerical analysis is that each thread is analyzed in isolation, then, the results of the thread-local analysis are propagated to all other threads
- Consider the analysis of this two threaded program
- If we first analyze thread one in isolation we can see it has a store on the shared variable x with the value of 5
- Similarly, analyzing thread two, we see a shared variable read of x. Since the thread is analyzed in isolation, thread two can only see the initial value of x which is zero
- Then, after analyzing each thread we propagate the values stored into shared memory across threads
- Specifically, we analyze thread two in the presense of the interfering store from thread one: this means thread two could either read the initial value or the value from thread one
- So, in thread two we determine the value of x to be between 0 and 5 which is sufficient to verify that the property that x is not equal to 10 is never violated

Thread Modular Numerical Abstract Interpretation

Initially: `x = 0`

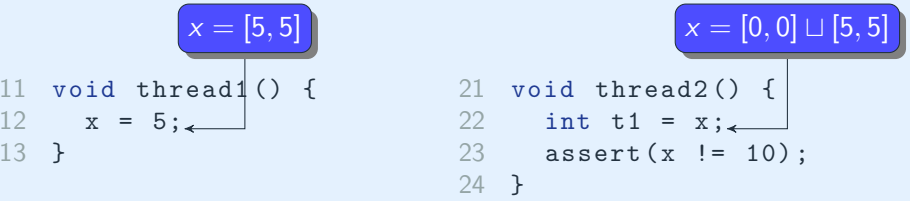


- Analyze threads in isolation
- Propagate *interfering* stores to loads

- Next, we'll look at how the previously described sequential analysis can be lifted to a thread-modular analysis
- The gist of the thread-modular numerical analysis is that each thread is analyzed in isolation, then, the results of the thread-local analysis are propagated to all other threads
- Consider the analysis of this two threaded program
- If we first analyze thread one in isolation we can see it has a store on the shared variable x with the value of 5
- Similarly, analyzing thread two, we see a shared variable read of x. Since the thread is analyzed in isolation, thread two can only see the initial value of x which is zero
- Then, after analyzing each thread we propagate the values stored into shared memory across threads
- Specifically, we analyze thread two in the presense of the interfering store from thread one: this means thread two could either read the initial value or the value from thread one
- So, in thread two we determine the value of x to be between 0 and 5 which is sufficient to verify that the property that x is not equal to 10 is never violated

Thread Modular Numerical Abstract Interpretation

Initially: `x = 0`

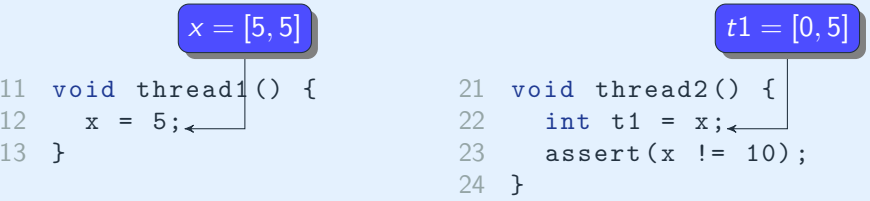


- Analyze threads in isolation
- Propagate *interfering* stores to loads

- Next, we'll look at how the previously described sequential analysis can be lifted to a thread-modular analysis
- The gist of the thread-modular numerical analysis is that each thread is analyzed in isolation, then, the results of the thread-local analysis are propagated to all other threads
- Consider the analysis of this two threaded program
- If we first analyze thread one in isolation we can see it has a store on the shared variable x with the value of 5
- Similarly, analyzing thread two, we see a shared variable read of x. Since the thread is analyzed in isolation, thread two can only see the initial value of x which is zero
- Then, after analyzing each thread we propagate the values stored into shared memory across threads
- Specifically, we analyze thread two in the presense of the interfering store from thread one: this means thread two could either read the initial value or the value from thread one
- So, in thread two we determine the value of x to be between 0 and 5 which is sufficient to verify that the property that x is not equal to 10 is never violated

Thread Modular Numerical Abstract Interpretation

Initially: `x = 0`



- Analyze threads in isolation
- Propagate *interfering* stores to loads

- Next, we'll look at how the previously described sequential analysis can be lifted to a thread-modular analysis
- The gist of the thread-modular numerical analysis is that each thread is analyzed in isolation, then, the results of the thread-local analysis are propagated to all other threads
- Consider the analysis of this two threaded program
- If we first analyze thread one in isolation we can see it has a store on the shared variable x with the value of 5
- Similarly, analyzing thread two, we see a shared variable read of x. Since the thread is analyzed in isolation, thread two can only see the initial value of x which is zero
- Then, after analyzing each thread we propagate the values stored into shared memory across threads
- Specifically, we analyze thread two in the presense of the interfering store from thread one: this means thread two could either read the initial value or the value from thread one
- So, in thread two we determine the value of x to be between 0 and 5 which is sufficient to verify that the property that x is not equal to 10 is never violated

Issues With Prior Work

Initially: `x = 0, flag = 0`

```
11 void thread1() {
12   x = 4;
13   x = 5;
14   flag = 1;
15 }
```

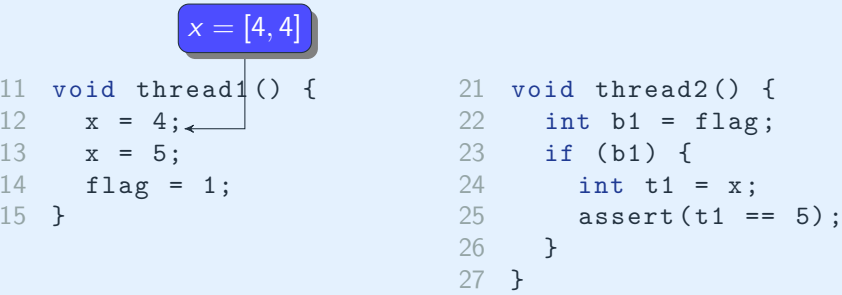
```
21 void thread2() {
22   int b1 = flag;
23   if (b1) {
24     int t1 = x;
25     assert(t1 == 5);
26   }
27 }
```

- Interference propagation is *flow-insensitive*

- However, prior work has accuracy issues since it handles the propagation of values across threads in a flow-insensitive manner: we can see this through the following example
- Consider this program where thread one performs only shared memory writes and thread two only reads
- First, the assertion in thread two is never violated: in order to reach the assertion, flag must be true to enter the first branch, which means that the write of x to 5 from thread one must have already occurred
- However, prior thread-modular analyses are unable to prove this condition
- Here's why: analyzing thread one produces three interfereces: x being equal to four or five, and flag being equal to true
- Then, analyzing thread two in the presence of these interferences means first that the value of flag is either zero, the initial value, or 1 from the interference
- Because flag is potentially true, we can enter the branch and perform the shared memory read on x
- Again, thread two can either read the initial value of zero or either 4 or 5 from thread one
- So, when the property is checked, we cannot show that the value of x must be 5:
- Since this property is never actually violated in the program, this is a false alarm.

Issues With Prior Work

Initially: `x = 0, flag = 0`

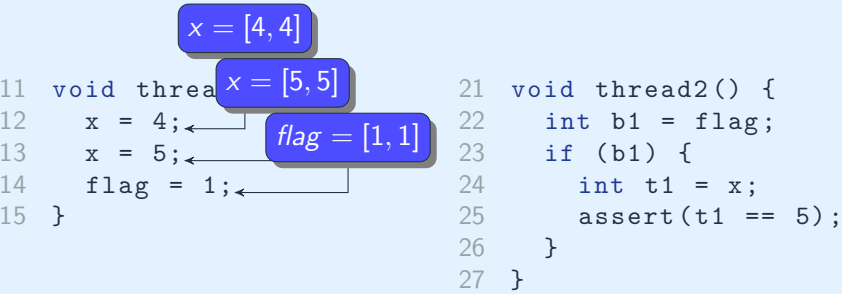


- Interference propagation is *flow-insensitive*

- However, prior work has accuracy issues since it handles the propagation of values across threads in a flow-insensitive manner: we can see this through the following example
- Consider this program where thread one performs only shared memory writes and thread two only reads
- First, the assertion in thread two is never violated: in order to reach the assertion, flag must be true to enter the first branch, which means that the write of x to 5 from thread one must have already occurred
- However, prior thread-modular analyses are unable to prove this condition
- Here's why: analyzing thread one produces three interfereces: x being equal to four or five, and flag being equal to true
- Then, analyzing thread two in the presence of these interferences means first that the value of flag is either zero, the initial value, or 1 from the interference
- Because flag is potentially true, we can enter the branch and perform the shared memory read on x
- Again, thread two can either read the initial value of zero or either 4 or 5 from thread one
- So, when the property is checked, we cannot show that the value of x must be 5:
- Since this property is never actually violated in the program, this is a false alarm.

Issues With Prior Work

Initially: `x = 0, flag = 0`

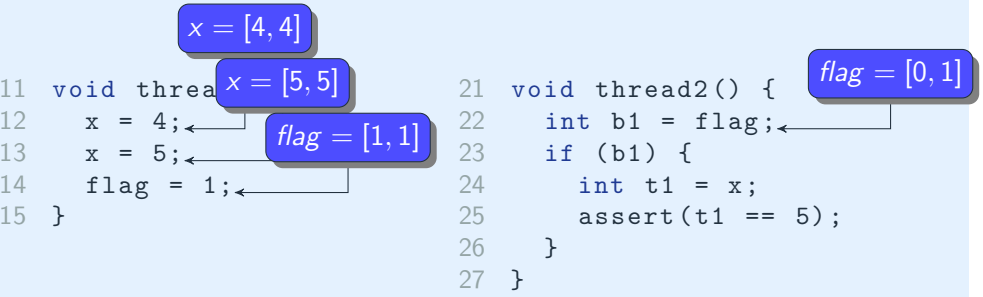


- Interference propagation is *flow-insensitive*

- However, prior work has accuracy issues since it handles the propagation of values across threads in a flow-insensitive manner: we can see this through the following example
- Consider this program where thread one performs only shared memory writes and thread two only reads
- First, the assertion in thread two is never violated: in order to reach the assertion, flag must be true to enter the first branch, which means that the write of x to 5 from thread one must have already occurred
- However, prior thread-modular analyses are unable to prove this condition
- Here's why: analyzing thread one produces three interfereces: x being equal to four or five, and flag being equal to true
- Then, analyzing thread two in the presence of these interferences means first that the value of flag is either zero, the initial value, or 1 from the interference
- Because flag is potentially true, we can enter the branch and perform the shared memory read on x
- Again, thread two can either read the initial value of zero or either 4 or 5 from thread one
- So, when the property is checked, we cannot show that the value of x must be 5:
- Since this property is never actually violated in the program, this is a false alarm.

Issues With Prior Work

Initially: `x = 0, flag = 0`

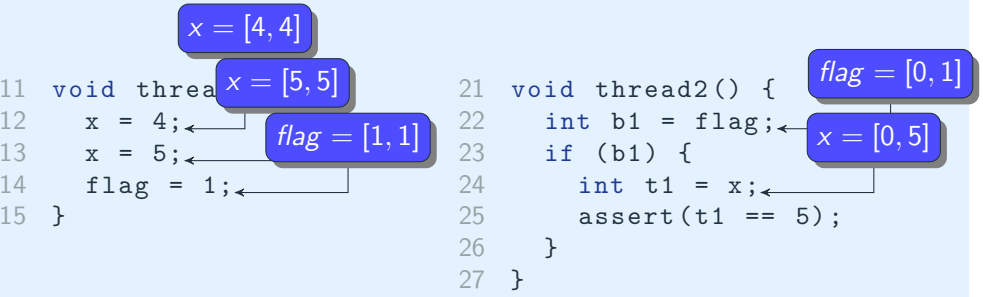


- Interference propagation is *flow-insensitive*

- However, prior work has accuracy issues since it handles the propagation of values across threads in a flow-insensitive manner: we can see this through the following example
- Consider this program where thread one performs only shared memory writes and thread two only reads
- First, the assertion in thread two is never violated: in order to reach the assertion, flag must be true to enter the first branch, which means that the write of x to 5 from thread one must have already occurred
- However, prior thread-modular analyses are unable to prove this condition
- Here's why: analyzing thread one produces three interfereces: x being equal to four or five, and flag being equal to true
- Then, analyzing thread two in the presence of these interferences means first that the value of flag is either zero, the initial value, or 1 from the interference
- Because flag is potentially true, we can enter the branch and perform the shared memory read on x
- Again, thread two can either read the initial value of zero or either 4 or 5 from thread one
- So, when the property is checked, we cannot show that the value of x must be 5:
- Since this property is never actually violated in the program, this is a false alarm.

Issues With Prior Work

Initially: `x = 0, flag = 0`

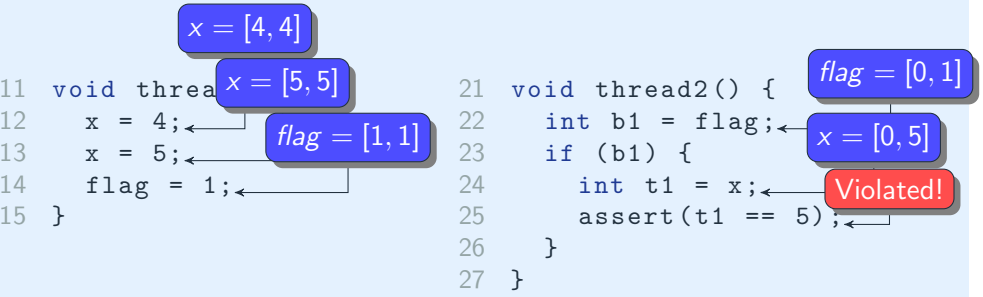


- Interference propagation is *flow-insensitive*

- However, prior work has accuracy issues since it handles the propagation of values across threads in a flow-insensitive manner: we can see this through the following example
- Consider this program where thread one performs only shared memory writes and thread two only reads
- First, the assertion in thread two is never violated: in order to reach the assertion, flag must be true to enter the first branch, which means that the write of x to 5 from thread one must have already occurred
- However, prior thread-modular analyses are unable to prove this condition
- Here's why: analyzing thread one produces three interfereces: x being equal to four or five, and flag being equal to true
- Then, analyzing thread two in the presence of these interferences means first that the value of flag is either zero, the initial value, or 1 from the interference
- Because flag is potentially true, we can enter the branch and perform the shared memory read on x
- Again, thread two can either read the initial value of zero or either 4 or 5 from thread one
- So, when the property is checked, we cannot show that the value of x must be 5:
- Since this property is never actually violated in the program, this is a false alarm.

Issues With Prior Work

Initially: `x = 0, flag = 0`



- Interference propagation is *flow-insensitive*

- However, prior work has accuracy issues since it handles the propagation of values across threads in a flow-insensitive manner: we can see this through the following example
- Consider this program where thread one performs only shared memory writes and thread two only reads
- First, the assertion in thread two is never violated: in order to reach the assertion, flag must be true to enter the first branch, which means that the write of x to 5 from thread one must have already occurred
- However, prior thread-modular analyses are unable to prove this condition
- Here's why: analyzing thread one produces three interfereces: x being equal to four or five, and flag being equal to true
- Then, analyzing thread two in the presence of these interferences means first that the value of flag is either zero, the initial value, or 1 from the interference
- Because flag is potentially true, we can enter the branch and perform the shared memory read on x
- Again, thread two can either read the initial value of zero or either 4 or 5 from thread one
- So, when the property is checked, we cannot show that the value of x must be 5:
- Since this property is never actually violated in the program, this is a false alarm.

Our New Approach

Initially: `x = 0, flag = 0`

```
11 void thread1() {
12   x = 4;
13   x = 5;
14   flag = 1;
15 }
```

```
21 void thread2() {
22   int b1 = flag;
23   if (b1) {
24     int t1 = x;
25     assert(t1 == 5);
26   }
27 }
```


- Our solution to this accuracy problem is two fold: first we delay joining together the values across all interferences, and second we use a system of constraints to prune infeasible interferences
- The combination of these two allow us to regain flow-sensitivity in the thread-modular analysis
- I'll re-analyze the previous example using our new technique to show both of these points
- First, if we consider analyzing thread two, there are six different combinations of interferences: x being 0, 4, or 5, and flag being either zero or one
- As we just saw, in prior work, all of these individual interferences would be joined together so x would be between zero and five, and flag would be zero or one
- By not joining all the values, we increase the acuraccy of the analysis by preventing the inclusion of spurious values
- Additionally, we use a system of constraints to check if all the possible interferences are actually feasible in the program
- Specifically, if we look at the six interferences, the first four do not violate the assertion since whenever flag is 1, x is equal to five
- For the remaining two interference combinations, our constraint analysis is able to show that they are infeasible in any possible concrete program execution

Our New Approach

Initially: `x = 0, flag = 0`

```
11 void thread1() {
12   x = 4;
13   x = 5;
14   flag = 1;
15 }
```

```
21 void thread2() {
22   int b1 = flag;
23   if (b1) {
24     int t1 = x;
25     assert(t1 == 5);
26   }
27 }
```

- Delay the join of interferences

- Our solution to this accuracy problem is two fold: first we delay joining together the values across all interferences, and second we use a system of constraints to prune infeasible interferences
- The combination of these two allow us to regain flow-sensitivity in the thread-modular analysis
- I'll re-analyze the previous example using our new technique to show both of these points
- First, if we consider analyzing thread two, there are six different combinations of interferences: x being 0, 4, or 5, and flag being either zero or one
- As we just saw, in prior work, all of these individual interferences would be joined together so x would be between zero and five, and flag would be zero or one
- By not joining all the values, we increase the acuraccy of the analysis by preventing the inclusion of spurious values
- Additionally, we use a system of constraints to check if all the possible interferences are actually feasible in the program
- Specifically, if we look at the six interferences, the first four do not violate the assertion since whenever flag is 1, x is equal to five
- For the remaining two interference combinations, our constraint analysis is able to show that they are infeasible in any possible concrete program execution

Our New Approach

Initially: `x = 0, flag = 0`

```
11 void thread1() {
12   x = 4;
13   x = 5;
14   flag = 1;
15 }
```

```
21 void thread2() {
22   int b1 = flag;
23   if (b1) {
24     int t1 = x;
25     assert(t1 == 5);
26   }
27 }
```

- Delay the join of interferences
- Use constraints to prune infeasible interferences

- Our solution to this accuracy problem is two fold: first we delay joining together the values across all interferences, and second we use a system of constraints to prune infeasible interferences
- The combination of these two allow us to regain flow-sensitivity in the thread-modular analysis
- I'll re-analyze the previous example using our new technique to show both of these points
- First, if we consider analyzing thread two, there are six different combinations of interferences: x being 0, 4, or 5, and flag being either zero or one
- As we just saw, in prior work, all of these individual interferences would be joined together so x would be between zero and five, and flag would be zero or one
- By not joining all the values, we increase the acuraccy of the analysis by preventing the inclusion of spurious values
- Additionally, we use a system of constraints to check if all the possible interferences are actually feasible in the program
- Specifically, if we look at the six interferences, the first four do not violate the assertion since whenever flag is 1, x is equal to five
- For the remaining two interference combinations, our constraint analysis is able to show that they are infeasible in any possible concrete program execution

Our New Approach

Initially: `x = 0, flag = 0`

```
11 void thread1() {
12   x = 4;
13   x = 5;
14   flag = 1;
15 }
```

```
21 void thread2() {
22   int b1 = flag;
23   if (b1) {
24     int t1 = x;
25     assert(t1 == 5);
26   }
27 }
```

- $(x = 4 \wedge flag = 0)$;
- $(x = 5 \wedge flag = 0)$;
- $(x = 0 \wedge flag = 0)$;
- $(x = 5 \wedge flag = 1)$;
- $(x = 4 \wedge flag = 1)$; and
- $(x = 0 \wedge flag = 1)$.

- Our solution to this accuracy problem is two fold: first we delay joining together the values across all interferences, and second we use a system of constraints to prune infeasible interferences
- The combination of these two allow us to regain flow-sensitivity in the thread-modular analysis
- I'll re-analyze the previous example using our new technique to show both of these points
- First, if we consider analyzing thread two, there are six different combinations of interferences: x being 0, 4, or 5, and flag being either zero or one
- As we just saw, in prior work, all of these individual interferences would be joined together so x would be between zero and five, and flag would be zero or one
- By not joining all the values, we increase the acuraccy of the analysis by preventing the inclusion of spurious values
- Additionally, we use a system of constraints to check if all the possible interferences are actually feasible in the program
- Specifically, if we look at the six interferences, the first four do not violate the assertion since whenever flag is 1, x is equal to five
- For the remaining two interference combinations, our constraint analysis is able to show that they are infeasible in any possible concrete program execution

Our New Approach

Initially: `x = 0, flag = 0`

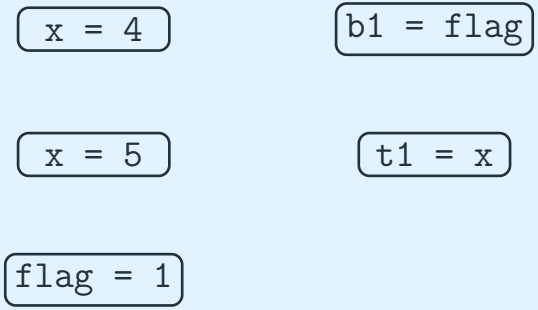
```
11 void thread1() {
12   x = 4;
13   x = 5;
14   flag = 1;
15 }
```

```
21 void thread2() {
22   int b1 = flag;
23   if (b1) {
24     int t1 = x;
25     assert(t1 == 5);
26   }
27 }
```

- $(x = 4 \wedge flag = 0)$;
- $(x = 5 \wedge flag = 0)$;
- $(x = 0 \wedge flag = 0)$;
- $(x = 5 \wedge flag = 1)$;
- $(x = 4 \wedge flag = 1)$; and
- $(x = 0 \wedge flag = 1)$.

- Our solution to this accuracy problem is two fold: first we delay joining together the values across all interferences, and second we use a system of constraints to prune infeasible interferences
- The combination of these two allow us to regain flow-sensitivity in the thread-modular analysis
- I'll re-analyze the previous example using our new technique to show both of these points
- First, if we consider analyzing thread two, there are six different combinations of interferences: x being 0, 4, or 5, and flag being either zero or one
- As we just saw, in prior work, all of these individual interferences would be joined together so x would be between zero and five, and flag would be zero or one
- By not joining all the values, we increase the acuraccy of the analysis by preventing the inclusion of spurious values
- Additionally, we use a system of constraints to check if all the possible interferences are actually feasible in the program
- Specifically, if we look at the six interferences, the first four do not violate the assertion since whenever flag is 1, x is equal to five
- For the remaining two interference combinations, our constraint analysis is able to show that they are infeasible in any possible concrete program execution

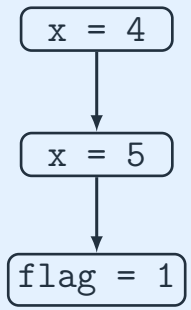
Constraint Analysis

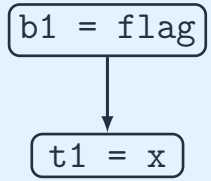


- $(x = 4 \wedge flag = 1)$

- Next, I'll provide an intuition as to how our constraint analysis is able to determine if an interference combination is infeasible
- We'll look at one of the interference combinations and show the constraints our analysis generates; at a high level, we use the program-order constraints within and across threads
- Again, here we have the same program with thread one on the left and thread two on the right
- The first constraints we consider are the program-order constraints within each thread: the execution order of statements within a thread must be consistent
- Next, we add constraints showing the flow of data across threads: since x reads the value of 4 from thread one, there is a flow of data from x equal to four to the subsequent read, and similarly for the read of flag
- Given these program-order constraints and constraints showing the flow of data, our analysis deduces additional implied ordering constraints
- Specifically, since the value of x is four, this means that thread two must have read x before thread ones subsequent write; otherwise, the value of x would have been overwritten
- Additionally, we can apply the transitive property of the ordering to deduce that the read of flag must have ocurred before the write
- Finally, if we examine the entire system of constraints, we can see that they form a contradiction: the read of flag must ocurr before the write, but also the flow of data must be from the write to the read

Constraint Analysis

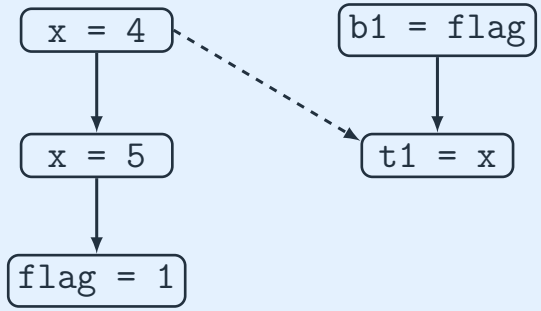




- $(x = 4 \wedge flag = 1)$

- Next, I'll provide an intuition as to how our constraint analysis is able to determine if an interference combination is infeasible
- We'll look at one of the interference combinations and show the constraints our analysis generates; at a high level, we use the program-order constraints within and across threads
- Again, here we have the same program with thread one on the left and thread two on the right
- The first constraints we consider are the program-order constraints within each thread: the execution order of statements within a thread must be consistent
- Next, we add constraints showing the flow of data across threads: since x reads the value of 4 from thread one, there is a flow of data from x equal to four to the subsequent read, and similarly for the read of flag
- Given these program-order constraints and constraints showing the flow of data, our analysis deduces additional implied ordering constraints
- Specifically, since the value of x is four, this means that thread two must have read x before thread ones subsequent write; otherwise, the value of x would have been overwritten
- Additionally, we can apply the transitive property of the ordering to deduce that the read of flag must have ocurred before the write
- Finally, if we examine the entire system of constraints, we can see that they form a contradiction: the read of flag must ocurr before the write, but also the flow of data must be from the write to the read

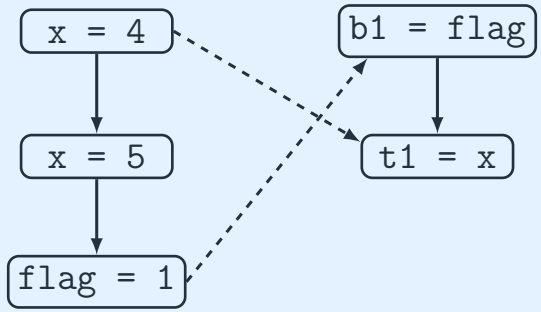
Constraint Analysis



- $(x = 4 \wedge flag = 1)$

- Next, I'll provide an intuition as to how our constraint analysis is able to determine if an interference combination is infeasible
- We'll look at one of the interference combinations and show the constraints our analysis generates; at a high level, we use the program-order constraints within and across threads
- Again, here we have the same program with thread one on the left and thread two on the right
- The first constraints we consider are the program-order constraints within each thread: the execution order of statements within a thread must be consistent
- Next, we add constraints showing the flow of data across threads: since x reads the value of 4 from thread one, there is a flow of data from x equal to four to the subsequent read, and similarly for the read of flag
- Given these program-order constraints and constraints showing the flow of data, our analysis deduces additional implied ordering constraints
- Specifically, since the value of x is four, this means that thread two must have read x before thread ones subsequent write; otherwise, the value of x would have been overwritten
- Additionally, we can apply the transitive property of the ordering to deduce that the read of flag must have ocurred before the write
- Finally, if we examine the entire system of constraints, we can see that they form a contradiction: the read of flag must ocurr before the write, but also the flow of data must be from the write to the read

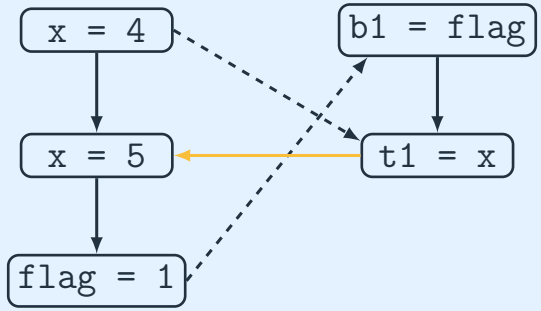
Constraint Analysis



- $(x = 4 \wedge flag = 1)$

- Next, I'll provide an intuition as to how our constraint analysis is able to determine if an interference combination is infeasible
- We'll look at one of the interference combinations and show the constraints our analysis generates; at a high level, we use the program-order constraints within and across threads
- Again, here we have the same program with thread one on the left and thread two on the right
- The first constraints we consider are the program-order constraints within each thread: the execution order of statements within a thread must be consistent
- Next, we add constraints showing the flow of data across threads: since x reads the value of 4 from thread one, there is a flow of data from x equal to four to the subsequent read, and similarly for the read of flag
- Given these program-order constraints and constraints showing the flow of data, our analysis deduces additional implied ordering constraints
- Specifically, since the value of x is four, this means that thread two must have read x before thread ones subsequent write; otherwise, the value of x would have been overwritten
- Additionally, we can apply the transitive property of the ordering to deduce that the read of flag must have ocurred before the write
- Finally, if we examine the entire system of constraints, we can see that they form a contradiction: the read of flag must ocurr before the write, but also the flow of data must be from the write to the read

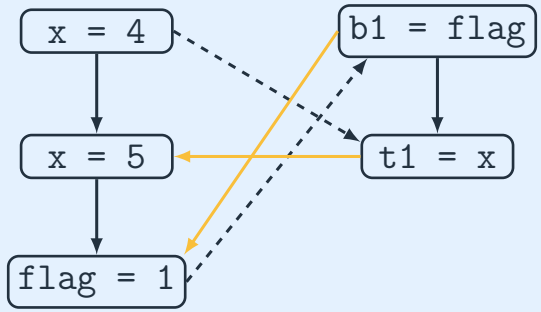
Constraint Analysis



- $(x = 4 \wedge flag = 1)$

- Next, I'll provide an intuition as to how our constraint analysis is able to determine if an interference combination is infeasible
- We'll look at one of the interference combinations and show the constraints our analysis generates; at a high level, we use the program-order constraints within and across threads
- Again, here we have the same program with thread one on the left and thread two on the right
- The first constraints we consider are the program-order constraints within each thread: the execution order of statements within a thread must be consistent
- Next, we add constraints showing the flow of data across threads: since x reads the value of 4 from thread one, there is a flow of data from x equal to four to the subsequent read, and similarly for the read of flag
- Given these program-order constraints and constraints showing the flow of data, our analysis deduces additional implied ordering constraints
- Specifically, since the value of x is four, this means that thread two must have read x before thread ones subsequent write; otherwise, the value of x would have been overwritten
- Additionally, we can apply the transitive property of the ordering to deduce that the read of flag must have ocurred before the write
- Finally, if we examine the entire system of constraints, we can see that they form a contradiction: the read of flag must ocurr before the write, but also the flow of data must be from the write to the read

Constraint Analysis



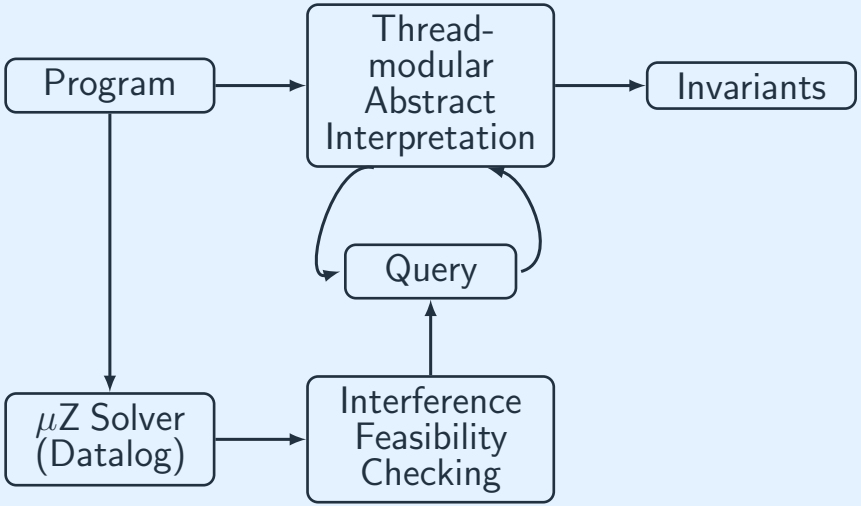
- $(x = 4 \wedge flag = 1)$

- Next, I'll provide an intuition as to how our constraint analysis is able to determine if an interference combination is infeasible
- We'll look at one of the interference combinations and show the constraints our analysis generates; at a high level, we use the program-order constraints within and across threads
- Again, here we have the same program with thread one on the left and thread two on the right
- The first constraints we consider are the program-order constraints within each thread: the execution order of statements within a thread must be consistent
- Next, we add constraints showing the flow of data across threads: since x reads the value of 4 from thread one, there is a flow of data from x equal to four to the subsequent read, and similarly for the read of flag
- Given these program-order constraints and constraints showing the flow of data, our analysis deduces additional implied ordering constraints
- Specifically, since the value of x is four, this means that thread two must have read x before thread ones subsequent write; otherwise, the value of x would have been overwritten
- Additionally, we can apply the transitive property of the ordering to deduce that the read of flag must have ocurred before the write
- Finally, if we examine the entire system of constraints, we can see that they form a contradiction: the read of flag must ocurr before the write, but also the flow of data must be from the write to the read

Overview

Now that I've gone over some examples, I'll provide the details of our new flow-sensitive analysis

Overview



- Here is a high-level overview of our new method
- We start with an input program we'd like to analyze
- We first perform an analysis to generate a set of constraints from the program which are fed into our constraint solver, which is a datalog solver
- Then, we perform a thread-modular analysis of each thread in the program
- During the thread modular analysis, as I just showed, each interference combination is analyzed in isolation
- Additionally, the abstract interpreter queries the solver to check if a certain interference combination is actually feasible
- If it is feasible, then it is analyzed, while if it is determined infeasible the interference can be safely skipped
- The end result is that each location in the program is annotated with a set of numerical invariants
- These invariants can then be used for things like property verification

Flow-sensitive Thread-modular Abstract Interpretation

```
11 void thread1() {
12   x = 4;
13   x = 5;
14   flag = 1;
15 }
```

```
21 void thread2() {
22   int b1 = flag;
23   if (b1) {
24     int t1 = x;
25     assert(t1 == 5);
26   }
27 }
```

- Interferences from thread one:

- Next, we'll look into closer detail of our flow-sensitive and thread-modular abstract interpretation; to do so, we'll again look at the same running example
- The gist of our technique is to explicitly test the cartesian product of all feasible store-to-load interferences
- First, we can see there are three interfering stores from thread one: two on x, setting it to four and give, and one on flag
- Similarly, there are two loads in thread two: one on flag and another on x
- Next, we pair each each load with every store to the same variable. This explicitly shows all the possible inter-thread data flows
- In addition to this, we have to consider the thread not reading from values from other threads but also reading values from itself. To this end, we use a special store s_d to represent the thread reading from its own local memory
- Next, we take the Cartesian product of all these sets which enumerates all the possible pairings of every load to a single store: these are the six possible combinations we looked at in the previous example
- Given these each of these combinations, we next check if each one individually is feasible in the program

Flow-sensitive Thread-modular Abstract Interpretation

```
11 void thread1() {
12   x = 4;
13   x = 5;
14   flag = 1;
15 }
```

```
21 void thread2() {
22   int b1 = flag;
23   if (b1) {
24     int t1 = x;
25     assert(t1 == 5);
26   }
27 }
```

- Interferences from thread one:
  - s_{x_1} : x = 4

- Next, we'll look into closer detail of our flow-sensitive and thread-modular abstract interpretation; to do so, we'll again look at the same running example
- The gist of our technique is to explicitly test the cartesian product of all feasible store-to-load interferences
- First, we can see there are three interfering stores from thread one: two on x, setting it to four and give, and one on flag
- Similarly, there are two loads in thread two: one on flag and another on x
- Next, we pair each each load with every store to the same variable. This explicitly shows all the possible inter-thread data flows
- In addition to this, we have to consider the thread not reading from values from other threads but also reading values from itself. To this end, we use a special store s_d to represent the thread reading from its own local memory
- Next, we take the Cartesian product of all these sets which enumerates all the possible pairings of every load to a single store: these are the six possible combinations we looked at in the previous example
- Given these each of these combinations, we next check if each one individually is feasible in the program

Flow-sensitive Thread-modular Abstract Interpretation

```
11 void thread1() {
12   x = 4;
13   x = 5;
14   flag = 1;
15 }
```

```
21 void thread2() {
22   int b1 = flag;
23   if (b1) {
24     int t1 = x;
25     assert(t1 == 5);
26   }
27 }
```

- Interferences from thread one:
  - s_{x_1} : `x = 4`
  - s_{x_2} : `x = 5`

- Next, we'll look into closer detail of our flow-sensitive and thread-modular abstract interpretation; to do so, we'll again look at the same running example
- The gist of our technique is to explicitly test the cartesian product of all feasible store-to-load interferences
- First, we can see there are three interfering stores from thread one: two on x, setting it to four and give, and one on flag
- Similarly, there are two loads in thread two: one on flag and another on x
- Next, we pair each each load with every store to the same variable. This explicitly shows all the possible inter-thread data flows
- In addition to this, we have to consider the thread not reading from values from other threads but also reading values from itself. To this end, we use a special store s_d to represent the thread reading from its own local memory
- Next, we take the Cartesian product of all these sets which enumerates all the possible pairings of every load to a single store: these are the six possible combinations we looked at in the previous example
- Given these each of these combinations, we next check if each one individually is feasible in the program

Flow-sensitive Thread-modular Abstract Interpretation

```
11 void thread1() {
12   x = 4;
13   x = 5;
14   flag = 1;
15 }
```

```
21 void thread2() {
22   int b1 = flag;
23   if (b1) {
24     int t1 = x;
25     assert(t1 == 5);
26   }
27 }
```

- Interferences from thread one:
  - s_{x_1} : `x = 4`
  - s_{x_2} : `x = 5`
  - s_f : `flag = 1`

- Next, we'll look into closer detail of our flow-sensitive and thread-modular abstract interpretation; to do so, we'll again look at the same running example
- The gist of our technique is to explicitly test the cartesian product of all feasible store-to-load interferences
- First, we can see there are three interfering stores from thread one: two on x, setting it to four and give, and one on flag
- Similarly, there are two loads in thread two: one on flag and another on x
- Next, we pair each each load with every store to the same variable. This explicitly shows all the possible inter-thread data flows
- In addition to this, we have to consider the thread not reading from values from other threads but also reading values from itself. To this end, we use a special store s_d to represent the thread reading from its own local memory
- Next, we take the Cartesian product of all these sets which enumerates all the possible pairings of every load to a single store: these are the six possible combinations we looked at in the previous example
- Given these each of these combinations, we next check if each one individually is feasible in the program

Flow-sensitive Thread-modular Abstract Interpretation

```
11 void thread1() {
12   x = 4;
13   x = 5;
14   flag = 1;
15 }
```

```
21 void thread2() {
22   int b1 = flag;
23   if (b1) {
24     int t1 = x;
25     assert(t1 == 5);
26   }
27 }
```

- Interferences from thread one:
  - s_{x_1} : `x = 4`
  - s_{x_2} : `x = 5`
  - s_f : `flag = 1`
- Loads from thread two:

- Next, we'll look into closer detail of our flow-sensitive and thread-modular abstract interpretation; to do so, we'll again look at the same running example
- The gist of our technique is to explicitly test the cartesian product of all feasible store-to-load interferences
- First, we can see there are three interfering stores from thread one: two on x, setting it to four and give, and one on flag
- Similarly, there are two loads in thread two: one on flag and another on x
- Next, we pair each each load with every store to the same variable. This explicitly shows all the possible inter-thread data flows
- In addition to this, we have to consider the thread not reading from values from other threads but also reading values from itself. To this end, we use a special store s_d to represent the thread reading from its own local memory
- Next, we take the Cartesian product of all these sets which enumerates all the possible pairings of every load to a single store: these are the six possible combinations we looked at in the previous example
- Given these each of these combinations, we next check if each one individually is feasible in the program

Flow-sensitive Thread-modular Abstract Interpretation

```
11 void thread1() {
12   x = 4;
13   x = 5;
14   flag = 1;
15 }
```

```
21 void thread2() {
22   int b1 = flag;
23   if (b1) {
24     int t1 = x;
25     assert(t1 == 5);
26   }
27 }
```

- Interferences from thread one:
  - s_{x_1} : `x = 4`
  - s_{x_2} : `x = 5`
  - s_f : `flag = 1`
- Loads from thread two:
  - l_f

- Next, we'll look into closer detail of our flow-sensitive and thread-modular abstract interpretation; to do so, we'll again look at the same running example
- The gist of our technique is to explicitly test the cartesian product of all feasible store-to-load interferences
- First, we can see there are three interfering stores from thread one: two on x, setting it to four and give, and one on flag
- Similarly, there are two loads in thread two: one on flag and another on x
- Next, we pair each each load with every store to the same variable. This explicitly shows all the possible inter-thread data flows
- In addition to this, we have to consider the thread not reading from values from other threads but also reading values from itself. To this end, we use a special store s_d to represent the thread reading from its own local memory
- Next, we take the Cartesian product of all these sets which enumerates all the possible pairings of every load to a single store: these are the six possible combinations we looked at in the previous example
- Given these each of these combinations, we next check if each one individually is feasible in the program

Flow-sensitive Thread-modular Abstract Interpretation

```
11 void thread1() {
12   x = 4;
13   x = 5;
14   flag = 1;
15 }
```

```
21 void thread2() {
22   int b1 = flag;
23   if (b1) {
24     int t1 = x;
25     assert(t1 == 5);
26   }
27 }
```

- Interferences from thread one:
  - s_{x_1} : `x = 4`
  - s_{x_2} : `x = 5`
  - s_f : `flag = 1`
- Loads from thread two:
  - l_f
  - l_x

- Next, we'll look into closer detail of our flow-sensitive and thread-modular abstract interpretation; to do so, we'll again look at the same running example
- The gist of our technique is to explicitly test the cartesian product of all feasible store-to-load interferences
- First, we can see there are three interfering stores from thread one: two on x, setting it to four and give, and one on flag
- Similarly, there are two loads in thread two: one on flag and another on x
- Next, we pair each each load with every store to the same variable. This explicitly shows all the possible inter-thread data flows
- In addition to this, we have to consider the thread not reading from values from other threads but also reading values from itself. To this end, we use a special store s_d to represent the thread reading from its own local memory
- Next, we take the Cartesian product of all these sets which enumerates all the possible pairings of every load to a single store: these are the six possible combinations we looked at in the previous example
- Given these each of these combinations, we next check if each one individually is feasible in the program

Flow-sensitive Thread-modular Abstract Interpretation

```
11 void thread1() {
12   x = 4;
13   x = 5;
14   flag = 1;
15 }
```

```
21 void thread2() {
22   int b1 = flag;
23   if (b1) {
24     int t1 = x;
25     assert(t1 == 5);
26   }
27 }
```

- Store-load-pairs: $(l_x, \{ s_{x_1}, s_{x_2} \}), (l_f, \{ s_f \})$

- Next, we'll look into closer detail of our flow-sensitive and thread-modular abstract interpretation; to do so, we'll again look at the same running example
- The gist of our technique is to explicitly test the cartesian product of all feasible store-to-load interferences
- First, we can see there are three interfering stores from thread one: two on x, setting it to four and give, and one on flag
- Similarly, there are two loads in thread two: one on flag and another on x
- Next, we pair each each load with every store to the same variable. This explicitly shows all the possible inter-thread data flows
- In addition to this, we have to consider the thread not reading from values from other threads but also reading values from itself. To this end, we use a special store s_d to represent the thread reading from its own local memory
- Next, we take the Cartesian product of all these sets which enumerates all the possible pairings of every load to a single store: these are the six possible combinations we looked at in the previous example
- Given these each of these combinations, we next check if each one individually is feasible in the program

Flow-sensitive Thread-modular Abstract Interpretation

```
11 void thread1() {
12   x = 4;
13   x = 5;
14   flag = 1;
15 }
```

```
21 void thread2() {
22   int b1 = flag;
23   if (b1) {
24     int t1 = x;
25     assert(t1 == 5);
26   }
27 }
```

- Load-store-pairs: $(l_x, \{ s_{x_1}, s_{x_2}, s_d \}), (l_f, \{ s_f, s_d \})$

- Next, we'll look into closer detail of our flow-sensitive and thread-modular abstract interpretation; to do so, we'll again look at the same running example
- The gist of our technique is to explicitly test the cartesian product of all feasible store-to-load interferences
- First, we can see there are three interfering stores from thread one: two on x, setting it to four and give, and one on flag
- Similarly, there are two loads in thread two: one on flag and another on x
- Next, we pair each each load with every store to the same variable. This explicitly shows all the possible inter-thread data flows
- In addition to this, we have to consider the thread not reading from values from other threads but also reading values from itself. To this end, we use a special store s_d to represent the thread reading from its own local memory
- Next, we take the Cartesian product of all these sets which enumerates all the possible pairings of every load to a single store: these are the six possible combinations we looked at in the previous example
- Given these each of these combinations, we next check if each one individually is feasible in the program

Flow-sensitive Thread-modular Abstract Interpretation

```
11 void thread1() {
12   x = 4;
13   x = 5;
14   flag = 1;
15 }
```

```
21 void thread2() {
22   int b1 = flag;
23   if (b1) {
24     int t1 = x;
25     assert(t1 == 5);
26   }
27 }
```

- Load-store-pairs: $(l_x, \{ s_{x_1}, s_{x_2}, s_d \}), (l_f, \{ s_f, s_d \})$
- Interference Combinations:
  - $(\langle l_x, s_{x_1} \rangle, \langle l_f, s_f \rangle)$
  - $(\langle l_x, s_{x_1} \rangle, \langle l_f, s_d \rangle)$
  - $(\langle l_x, s_{x_2} \rangle, \langle l_f, s_f \rangle)$
  - $(\langle l_x, s_{x_2} \rangle, \langle l_f, s_d \rangle)$
  - $(\langle l_x, s_d \rangle, \langle l_f, s_f \rangle)$
  - $(\langle l_x, s_d \rangle, \langle l_f, s_d \rangle)$

- Next, we'll look into closer detail of our flow-sensitive and thread-modular abstract interpretation; to do so, we'll again look at the same running example
- The gist of our technique is to explicitly test the cartesian product of all feasible store-to-load interferences
- First, we can see there are three interfering stores from thread one: two on x, setting it to four and give, and one on flag
- Similarly, there are two loads in thread two: one on flag and another on x
- Next, we pair each each load with every store to the same variable. This explicitly shows all the possible inter-thread data flows
- In addition to this, we have to consider the thread not reading from values from other threads but also reading values from itself. To this end, we use a special store s_d to represent the thread reading from its own local memory
- Next, we take the Cartesian product of all these sets which enumerates all the possible pairings of every load to a single store: these are the six possible combinations we looked at in the previous example
- Given these each of these combinations, we next check if each one individually is feasible in the program

Checking Interference Feasibility

- Horn-clauses with finite domains

- To check the feasibility of interferences, we generate and check the feasibility of a set of horn clauses over finite domains
- We used horn-clauses, as opposed to other constraints like SMT or SAT, because they have efficient decision procedures: satisfiability can be checked in polynomial time
- The main goal of our analysis is to determine which store-to-load flows are infeasible; we do this mostly by generating the must-happen-before relation; this relation indicates that two statements must have a specific ordering
- One of the keys to making our analysis accurate is that we only generate the must-happen-before relation with respect to a single interference combination as opposed to over the entire program
- Finally, we also ensure our constraint analysis is sound: to do this, we ensure that we under-approximate the must-happen-before relation
- This means that whenever we determine an interference combination as infeasible, it definitely is infeasible in the actual program
- However, we do not guarantee that all infeasible interference combinations will be detected

Checking Interference Feasibility

- Horn-clauses with finite domains
  - Solved in polynomial time

- To check the feasibility of interferences, we generate and check the feasibility of a set of horn clauses over finite domains
- We used horn-clauses, as opposed to other constraints like SMT or SAT, because they have efficient decision procedures: satisfiability can be checked in polynomial time
- The main goal of our analysis is to determine which store-to-load flows are infeasible; we do this mostly by generating the must-happen-before relation; this relation indicates that two statements must have a specific ordering
- One of the keys to making our analysis accurate is that we only generate the must-happen-before relation with respect to a single interference combination as opposed to over the entire program
- Finally, we also ensure our constraint analysis is sound: to do this, we ensure that we under-approximate the must-happen-before relation
- This means that whenever we determine an interference combination as infeasible, it definitely is infeasible in the actual program
- However, we do not guarantee that all infeasible interference combinations will be detected

Checking Interference Feasibility

- Horn-clauses with finite domains
  - Solved in polynomial time
- Must-happen-before relation

- To check the feasibility of interferences, we generate and check the feasibility of a set of horn clauses over finite domains
- We used horn-clauses, as opposed to other constraints like SMT or SAT, because they have efficient decision procedures: satisfiability can be checked in polynomial time
- The main goal of our analysis is to determine which store-to-load flows are infeasible; we do this mostly by generating the must-happen-before relation; this relation indicates that two statements must have a specific ordering
- One of the keys to making our analysis accurate is that we only generate the must-happen-before relation with respect to a single interference combination as opposed to over the entire program
- Finally, we also ensure our constraint analysis is sound: to do this, we ensure that we under-approximate the must-happen-before relation
- This means that whenever we determine an interference combination as infeasible, it definitely is infeasible in the actual program
- However, we do not guarantee that all infeasible interference combinations will be detected

Checking Interference Feasibility

- Horn-clauses with finite domains
  - Solved in polynomial time
- Must-happen-before relation
  - Only analyze a single interference combination

- To check the feasibility of interferences, we generate and check the feasibility of a set of horn clauses over finite domains
- We used horn-clauses, as opposed to other constraints like SMT or SAT, because they have efficient decision procedures: satisfiability can be checked in polynomial time
- The main goal of our analysis is to determine which store-to-load flows are infeasible; we do this mostly by generating the must-happen-before relation; this relation indicates that two statements must have a specific ordering
- One of the keys to making our analysis accurate is that we only generate the must-happen-before relation with respect to a single interference combination as opposed to over the entire program
- Finally, we also ensure our constraint analysis is sound: to do this, we ensure that we under-approximate the must-happen-before relation
- This means that whenever we determine an interference combination as infeasible, it definitely is infeasible in the actual program
- However, we do not guarantee that all infeasible interference combinations will be detected

Checking Interference Feasibility

- Horn-clauses with finite domains
  - Solved in polynomial time
- Must-happen-before relation
  - Only analyze a single interference combination
- Contradictions in must-happen-before indicate infeasibility

- To check the feasibility of interferences, we generate and check the feasibility of a set of horn clauses over finite domains
- We used horn-clauses, as opposed to other constraints like SMT or SAT, because they have efficient decision procedures: satisfiability can be checked in polynomial time
- The main goal of our analysis is to determine which store-to-load flows are infeasible; we do this mostly by generating the must-happen-before relation; this relation indicates that two statements must have a specific ordering
- One of the keys to making our analysis accurate is that we only generate the must-happen-before relation with respect to a single interference combination as opposed to over the entire program
- Finally, we also ensure our constraint analysis is sound: to do this, we ensure that we under-approximate the must-happen-before relation
- This means that whenever we determine an interference combination as infeasible, it definitely is infeasible in the actual program
- However, we do not guarantee that all infeasible interference combinations will be detected

Checking Interference Feasibility

- Horn-clauses with finite domains
  - Solved in polynomial time
- Must-happen-before relation
  - Only analyze a single interference combination
- Contradictions in must-happen-before indicate infeasibility
- Sound

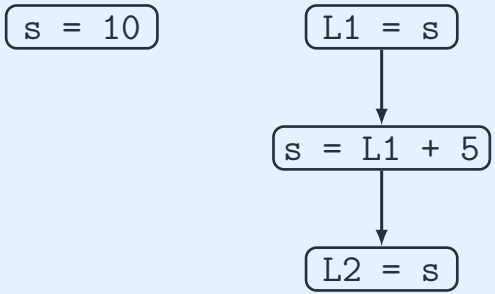
- To check the feasibility of interferences, we generate and check the feasibility of a set of horn clauses over finite domains
- We used horn-clauses, as opposed to other constraints like SMT or SAT, because they have efficient decision procedures: satisfiability can be checked in polynomial time
- The main goal of our analysis is to determine which store-to-load flows are infeasible; we do this mostly by generating the must-happen-before relation; this relation indicates that two statements must have a specific ordering
- One of the keys to making our analysis accurate is that we only generate the must-happen-before relation with respect to a single interference combination as opposed to over the entire program
- Finally, we also ensure our constraint analysis is sound: to do this, we ensure that we under-approximate the must-happen-before relation
- This means that whenever we determine an interference combination as infeasible, it definitely is infeasible in the actual program
- However, we do not guarantee that all infeasible interference combinations will be detected

Checking Interference Feasibility

- Horn-clauses with finite domains
  - Solved in polynomial time
- Must-happen-before relation
  - Only analyze a single interference combination
- Contradictions in must-happen-before indicate infeasibility
- Sound
  - Under-approximate must-happen-before

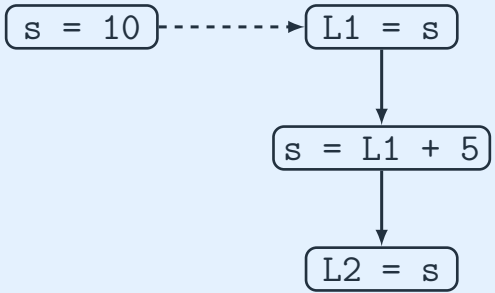
- To check the feasibility of interferences, we generate and check the feasibility of a set of horn clauses over finite domains
- We used horn-clauses, as opposed to other constraints like SMT or SAT, because they have efficient decision procedures: satisfiability can be checked in polynomial time
- The main goal of our analysis is to determine which store-to-load flows are infeasible; we do this mostly by generating the must-happen-before relation; this relation indicates that two statements must have a specific ordering
- One of the keys to making our analysis accurate is that we only generate the must-happen-before relation with respect to a single interference combination as opposed to over the entire program
- Finally, we also ensure our constraint analysis is sound: to do this, we ensure that we under-approximate the must-happen-before relation
- This means that whenever we determine an interference combination as infeasible, it definitely is infeasible in the actual program
- However, we do not guarantee that all infeasible interference combinations will be detected

Deduction Rules



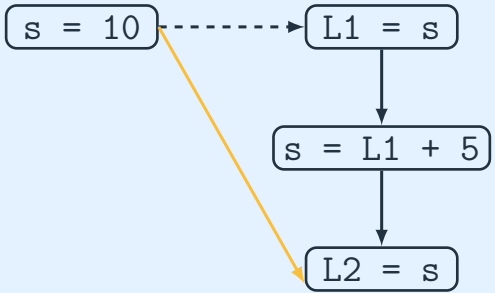
- We created a bunch of deduction rules capturing various scenarios where a must-happen-before edge can be inferred
- Here's an example of one of them. We have one thread on the left performing a store, and a thread on the right performing a load, store, and the subsequent load
- The program-order constraints within each thread are represented by the black arrows
- Suppose that the interference combination we are analyzing contains a store-to-load flow as shown by the dashed edge
- Notice that the right thread is first performing a shared memory read and then performing a shared memory write on the same variable
- This means that the subsequent memory read cannot read the same value: this is because the thread has already overwritten it
- So, we deduce that given the first store-to-load flow, the second yellow store-to-load flow of the right thread is infeasible
- This means that any interference combination containing this infeasible store-to-load combination can be removed from the analysis
- We created other deduction rules in a similar fashion by examining other situations where inter-thread flows become infeasible

Deduction Rules



- We created a bunch of deduction rules capturing various scenarios where a must-happen-before edge can be inferred
- Here's an example of one of them. We have one thread on the left performing a store, and a thread on the right performing a load, store, and the subsequent load
- The program-order constraints within each thread are represented by the black arrows
- Suppose that the interference combination we are analyzing contains a store-to-load flow as shown by the dashed edge
- Notice that the right thread is first performing a shared memory read and then performing a shared memory write on the same variable
- This means that the subsequent memory read cannot read the same value: this is because the thread has already overwritten it
- So, we deduce that given the first store-to-load flow, the second yellow store-to-load flow of the right thread is infeasible
- This means that any interference combination containing this infeasible store-to-load combination can be removed from the analysis
- We created other deduction rules in a similar fashion by examining other situations where inter-thread flows become infeasible

Deduction Rules



- We created a bunch of deduction rules capturing various scenarios where a must-happen-before edge can be inferred
- Here's an example of one of them. We have one thread on the left performing a store, and a thread on the right performing a load, store, and the subsequent load
- The program-order constraints within each thread are represented by the black arrows
- Suppose that the interference combination we are analyzing contains a store-to-load flow as shown by the dashed edge
- Notice that the right thread is first performing a shared memory read and then performing a shared memory write on the same variable
- This means that the subsequent memory read cannot read the same value: this is because the thread has already overwritten it
- So, we deduce that given the first store-to-load flow, the second yellow store-to-load flow of the right thread is infeasible
- This means that any interference combination containing this infeasible store-to-load combination can be removed from the analysis
- We created other deduction rules in a similar fashion by examining other situations where inter-thread flows become infeasible

Handling Loops

```
int x = 0;
void thread1() {
  create(thread2);
  while (*) {
    int t1 = x;
  }
  create(thread3);
}
```

```
void thread2() {
  x = 1;
  x = 2;
}
void thread3() {
  x = 10;
}
```


- Thus far, I've ignored the case where shared memory reads are performed within a loop
- In general, when a load is in a loop it can read from an arbitrary, and potentially infinite, number of different interferences
- As a result, if we simply treat loads within loops in the same way as as previously described there could be an infinite number of infinitely long combinations
- Consider this example, thread one creates thread two and then repeatedly loads the value of x an arbitrary number of times before creating thread two
- Notice that there is a must-happen-before edges between the load within the loop and the write to x by thread 3: because thread 3 is created after the load, the load cannot read the values written by thread 3
- So, the value read within the loop could be any sequence starting with an arbitrary number of zeros, followed by ones, followed by 2s
- Our previous system of constraints cannot handle such an infinite sequence so, instead, we merge all the potential interferences into a single value
- This means that the loads performed across loop-iterations will not be handled in a flow-sensitive way, but, any interfering store which cannot be read on any iteration will be remove thus improving acurracy

Handling Loops

```
int x = 0;
void thread1() {
  create(thread2);
  while (*) {
    int t1 = x;
  }
  create(thread3);
}
```

```
void thread2() {
  x = 1;
  x = 2;
}
void thread3() {
  x = 10;
}
```

- 0*1*2*

- Thus far, I've ignored the case where shared memory reads are performed within a loop
- In general, when a load is in a loop it can read from an arbitrary, and potentially infinite, number of different interferences
- As a result, if we simply treat loads within loops in the same way as as previously described there could be an infinite number of infinitely long combinations
- Consider this example, thread one creates thread two and then repeatedly loads the value of x an arbitrary number of times before creating thread two
- Notice that there is a must-happen-before edges between the load within the loop and the write to x by thread 3: because thread 3 is created after the load, the load cannot read the values written by thread 3
- So, the value read within the loop could be any sequence starting with an arbitrary number of zeros, followed by ones, followed by 2s
- Our previous system of constraints cannot handle such an infinite sequence so, instead, we merge all the potential interferences into a single value
- This means that the loads performed across loop-iterations will not be handled in a flow-sensitive way, but, any interfering store which cannot be read on any iteration will be remove thus improving acurracy

Handling Loops

```
int x = 0;
void thread1() {
  create(thread2);
  while (*) {
    int t1 = x;
  }
  create(thread3);
}
```

```
void thread2() {
  x = 1;
  x = 2;
}
void thread3() {
  x = 10;
}
```

- 0*1*2*
- $\{0, 1, 2\}$

- Thus far, I've ignored the case where shared memory reads are performed within a loop
- In general, when a load is in a loop it can read from an arbitrary, and potentially infinite, number of different interferences
- As a result, if we simply treat loads within loops in the same way as as previously described there could be an infinite number of infinitely long combinations
- Consider this example, thread one creates thread two and then repeatedly loads the value of x an arbitrary number of times before creating thread two
- Notice that there is a must-happen-before edges between the load within the loop and the write to x by thread 3: because thread 3 is created after the load, the load cannot read the values written by thread 3
- So, the value read within the loop could be any sequence starting with an arbitrary number of zeros, followed by ones, followed by 2s
- Our previous system of constraints cannot handle such an infinite sequence so, instead, we merge all the potential interferences into a single value
- This means that the loads performed across loop-iterations will not be handled in a flow-sensitive way, but, any interfering store which cannot be read on any iteration will be remove thus improving acurracy

Optimizations

- Property directed pruning

- Next, I'll go over two optimizations we created for our technique
- The two optimizations we used we call property directed pruning and clustering

Optimizations

- Property directed pruning
- Clustering

- Next, I'll go over two optimizations we created for our technique
- The two optimizations we used we call property directed pruning and clustering

Property Directed Pruning

```
int x = 0;
void thr(int v) {
  int t1 = 5 * v;
  int t2 = x;
  x = t1 + t2;
  if (t1 < 0)
    ERROR!;
}
```

```
int main() {
  thread_create(thr,5);
  thread_create(thr,10);
  x = 1;
  thread_exit(0);
}
```


- First, property directed pruning uses slices from the program-dependence graph to remove redundant statements
- If we look at this program, the main thread, on the right, creates two threads with arguments 5 and 10, and then performs a shared memory write to x
- The threads use their argument to perform a computation, and then perform a separate computation related to x
- The property being checked is related to t1, and the input argument to the thread
- If we look at the property being checking within each thread, it only depends on the input to the thread: the shared memory read on x has no influence on the reachability of the ERROR
- So, during the abstract interpretation, we can skip those statements involved with x
- Additionally, when exploring possible interference combinations, we can skip those related to x
- So, this both reduces the number of interference combinations and also reduces the individual analysis of each thread
- These dependencies between statements are captured easily by using the program dependence graph

Clustering

```
int x = 0;
int y = 0;
void thread1() {
  x = 1;
  y = 1;
}
```

```
void thread2() {
  int t1 = x;
  int t2 = y;
  assert(x >= 0);
  assert(y >= 0);
}
```


- The second optimization we use is called clustering and also uses the program dependence graph
- This optimization reduces the number of possible interference combinations thereby reducing the runtime of the analysis
- Here we have two threads, one performing shared memory writes to x and y, and the other is performing shared memory read
- The two properties being analyzed are separately on x and y
- Using the previously described technique, there are two possible interfering stores for each of the loads in thread 2: one from thread 1 and one causing thread 2 to read from its own memory environment
- This means that thread 2 will have to be analyzed 4 times, one for each combination
- However, if we look at the program, the properties being checked are performed on disjoint computations
- Specifically, the writes to x and the writes to y do not influence each other
- So, during the analysis, the combinations between the reads of x and y in thread two can be performed independently
- As a result, we only need to consider the two combinations independently meaning we only need to run the analysis twice for thread two

Clustering

```
int x = 0;
int y = 0;
void thread1() {
  x = 1;
  y = 1;
}
```

```
void thread2() {
  int t1 = x;
  int t2 = y;
  assert(x >= 0);
  assert(y >= 0);
}
```

- $2 * 2 = 4$ interference combinations

- The second optimization we use is called clustering and also uses the program dependence graph
- This optimization reduces the number of possible interference combinations thereby reducing the runtime of the analysis
- Here we have two threads, one performing shared memory writes to x and y, and the other is performing shared memory read
- The two properties being analyzed are separately on x and y
- Using the previously described technique, there are two possible interfering stores for each of the loads in thread 2: one from thread 1 and one causing thread 2 to read from its own memory environment
- This means that thread 2 will have to be analyzed 4 times, one for each combination
- However, if we look at the program, the properties being checked are performed on disjoint computations
- Specifically, the writes to x and the writes to y do not influence each other
- So, during the analysis, the combinations between the reads of x and y in thread two can be performed independently
- As a result, we only need to consider the two combinations independently meaning we only need to run the analysis twice for thread two

Clustering

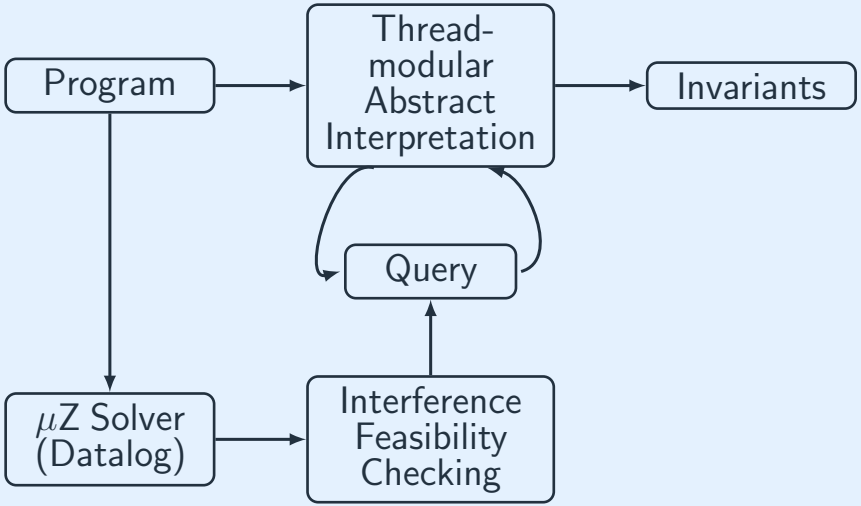
```
int x = 0;
int y = 0;
void thread1() {
  x = 1;
  y = 1;
}
```

```
void thread2() {
  int t1 = x;
  int t2 = y;
  assert(x >= 0);
  assert(y >= 0);
}
```

- $2 * 2 = 4$ interference combinations
- Clustering: 2 combinations

- The second optimization we use is called clustering and also uses the program dependence graph
- This optimization reduces the number of possible interference combinations thereby reducing the runtime of the analysis
- Here we have two threads, one performing shared memory writes to x and y, and the other is performing shared memory read
- The two properties being analyzed are separately on x and y
- Using the previously described technique, there are two possible interfering stores for each of the loads in thread 2: one from thread 1 and one causing thread 2 to read from its own memory environment
- This means that thread 2 will have to be analyzed 4 times, one for each combination
- However, if we look at the program, the properties being checked are performed on disjoint computations
- Specifically, the writes to x and the writes to y do not influence each other
- So, during the analysis, the combinations between the reads of x and y in thread two can be performed independently
- As a result, we only need to consider the two combinations independently meaning we only need to run the analysis twice for thread two

Overview



- At this point, I've gone over all the big points of our flow-sensitive analysis
- I first showed how we considered and calculated different interference combinations during the thread-modular abstract interpretation
- And second, I showed some examples of how we determine if certain interference combinations were infeasible
- By repeatedly applying these two steps, we generate a set of more accurate invariants, compared to prior work, for each thread

Soundness

- As sound as underlying sequential abstract interpreter

- Finally, I'll provide a discussion on the soundness of our approach
- Most of the thread-modular analysis builds upon the sequential abstract interpreter
- This is because each thread is, essentially, analyzed individually as a sequential program and then has its results propagated to other threads
- Our approach has its soundness based on this approach: we are just as sound as the underlying abstract interpreter
- This is because we largely are not modifying the semantics of the sequential interpreter
- Specifically, our consideration of interference combinations in isolation is a form of focusing
- At a high level, focusing takes the input to the analysis and instead of analyzing the entire input it breaks it up into smaller pieces
- Each of these pieces are analyzed individually and then the results are combined
- Prior work has already discussed the soundness of this technique and we build of these findings

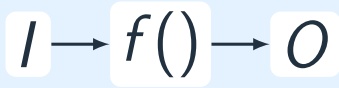
Soundness

- As sound as underlying sequential abstract interpreter
- Interference combinations *focus* the analysis

- Finally, I'll provide a discussion on the soundness of our approach
- Most of the thread-modular analysis builds upon the sequential abstract interpreter
- This is because each thread is, essentially, analyzed individually as a sequential program and then has its results propagated to other threads
- Our approach has its soundness based on this approach: we are just as sound as the underlying abstract interpreter
- This is because we largely are not modifying the semantics of the sequential interpreter
- Specifically, our consideration of interference combinations in isolation is a form of focusing
- At a high level, focusing takes the input to the analysis and instead of analyzing the entire input it breaks it up into smaller pieces
- Each of these pieces are analyzed individually and then the results are combined
- Prior work has already discussed the soundness of this technique and we build of these findings

Soundness

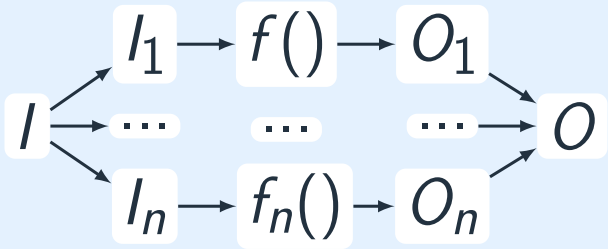
- As sound as underlying sequential abstract interpreter
- Interference combinations *focus* the analysis



- Finally, I'll provide a discussion on the soundness of our approach
- Most of the thread-modular analysis builds upon the sequential abstract interpreter
- This is because each thread is, essentially, analyzed individually as a sequential program and then has its results propagated to other threads
- Our approach has its soundness based on this approach: we are just as sound as the underlying abstract interpreter
- This is because we largely are not modifying the semantics of the sequential interpreter
- Specifically, our consideration of interference combinations in isolation is a form of focusing
- At a high level, focusing takes the input to the analysis and instead of analyzing the entire input it breaks it up into smaller pieces
- Each of these pieces are analyzed individually and then the results are combined
- Prior work has already discussed the soundness of this technique and we build of these findings

Soundness

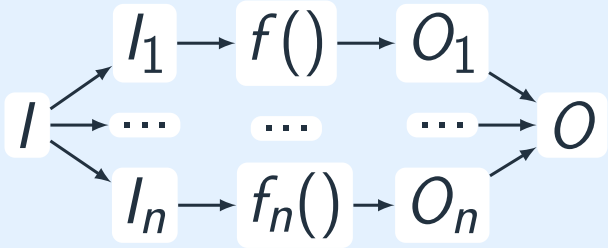
- As sound as underlying sequential abstract interpreter
- Interference combinations *focus* the analysis



- Finally, I'll provide a discussion on the soundness of our approach
- Most of the thread-modular analysis builds upon the sequential abstract interpreter
- This is because each thread is, essentially, analyzed individually as a sequential program and then has its results propagated to other threads
- Our approach has its soundness based on this approach: we are just as sound as the underlying abstract interpreter
- This is because we largely are not modifying the semantics of the sequential interpreter
- Specifically, our consideration of interference combinations in isolation is a form of focusing
- At a high level, focusing takes the input to the analysis and instead of analyzing the entire input it breaks it up into smaller pieces
- Each of these pieces are analyzed individually and then the results are combined
- Prior work has already discussed the soundness of this technique and we build of these findings

Soundness

- As sound as underlying sequential abstract interpreter
- Interference combinations *focus* the analysis
  - *Systematic Design of Program Analysis Frameworks.* Patrick Cousot and Radhia Cousot. POPL '79.
  - *Parametric Shape Analysis via 3-valued Logic.* Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. POPL '99, TOPLAS '02.



- Finally, I'll provide a discussion on the soundness of our approach
- Most of the thread-modular analysis builds upon the sequential abstract interpreter
- This is because each thread is, essentially, analyzed individually as a sequential program and then has its results propagated to other threads
- Our approach has its soundness based on this approach: we are just as sound as the underlying abstract interpreter
- This is because we largely are not modifying the semantics of the sequential interpreter
- Specifically, our consideration of interference combinations in isolation is a form of focusing
- At a high level, focusing takes the input to the analysis and instead of analyzing the entire input it breaks it up into smaller pieces
- Each of these pieces are analyzed individually and then the results are combined
- Prior work has already discussed the soundness of this technique and we build of these findings

Overview

Next, I'll go over our experimental results

Experimental Overview

- LLVM abstract interpreter

- We implemented prior techniques and our own using the LLVM compiler framework to create an abstract interpreter
- We used Apron for implementation of abstract domains and transfer functions
- And, to solve our constraints we used the mu-Z datalog engine inside Z3
- We evaluated our tool against prior work using 38 C programs
- These benchmarks were ones we wrote ourself as well as some from the Software Verification competition and linux device drivers
- All the tests were run on a 2.6 GHz processor with 8 GB RAM

Experimental Overview

- LLVM abstract interpreter
- Apron abstract domains

- We implemented prior techniques and our own using the LLVM compiler framework to create an abstract interpreter
- We used Apron for implementation of abstract domains and transfer functions
- And, to solve our constraints we used the mu-Z datalog engine inside Z3
- We evaluated our tool against prior work using 38 C programs
- These benchmarks were ones we wrote ourself as well as some from the Software Verification competition and linux device drivers
- All the tests were run on a 2.6 GHz processor with 8 GB RAM

Experimental Overview

- LLVM abstract interpreter
- Apron abstract domains
- μZ (Z3) Datalog solver

- We implemented prior techniques and our own using the LLVM compiler framework to create an abstract interpreter
- We used Apron for implementation of abstract domains and transfer functions
- And, to solve our constraints we used the mu-Z datalog engine inside Z3
- We evaluated our tool against prior work using 38 C programs
- These benchmarks were ones we wrote ourself as well as some from the Software Verification competition and linux device drivers
- All the tests were run on a 2.6 GHz processor with 8 GB RAM

Experimental Overview

- LLVM abstract interpreter
- Apron abstract domains
- μZ (Z3) Datalog solver
- 38 C programs

- We implemented prior techniques and our own using the LLVM compiler framework to create an abstract interpreter
- We used Apron for implementation of abstract domains and transfer functions
- And, to solve our constraints we used the mu-Z datalog engine inside Z3
- We evaluated our tool against prior work using 38 C programs
- These benchmarks were ones we wrote ourself as well as some from the Software Verification competition and linux device drivers
- All the tests were run on a 2.6 GHz processor with 8 GB RAM

Experimental Overview

- LLVM abstract interpreter
- Apron abstract domains
- μZ (Z3) Datalog solver
- 38 C programs
  - SVCOMP

- We implemented prior techniques and our own using the LLVM compiler framework to create an abstract interpreter
- We used Apron for implementation of abstract domains and transfer functions
- And, to solve our constraints we used the mu-Z datalog engine inside Z3
- We evaluated our tool against prior work using 38 C programs
- These benchmarks were ones we wrote ourself as well as some from the Software Verification competition and linux device drivers
- All the tests were run on a 2.6 GHz processor with 8 GB RAM

Experimental Overview

- LLVM abstract interpreter
- Apron abstract domains
- μZ (Z3) Datalog solver
- 38 C programs
  - SVCOMP
  - Linux device drivers

- We implemented prior techniques and our own using the LLVM compiler framework to create an abstract interpreter
- We used Apron for implementation of abstract domains and transfer functions
- And, to solve our constraints we used the mu-Z datalog engine inside Z3
- We evaluated our tool against prior work using 38 C programs
- These benchmarks were ones we wrote ourself as well as some from the Software Verification competition and linux device drivers
- All the tests were run on a 2.6 GHz processor with 8 GB RAM

Experimental Overview

- LLVM abstract interpreter
- Apron abstract domains
- μZ (Z3) Datalog solver
- 38 C programs
  - SVCOMP
  - Linux device drivers
- 2.6 GHz processor

- We implemented prior techniques and our own using the LLVM compiler framework to create an abstract interpreter
- We used Apron for implementation of abstract domains and transfer functions
- And, to solve our constraints we used the mu-Z datalog engine inside Z3
- We evaluated our tool against prior work using 38 C programs
- These benchmarks were ones we wrote ourself as well as some from the Software Verification competition and linux device drivers
- All the tests were run on a 2.6 GHz processor with 8 GB RAM

Experimental Overview

- LLVM abstract interpreter
- Apron abstract domains
- μZ (Z3) Datalog solver
- 38 C programs
  - SVCOMP
  - Linux device drivers
- 2.6 GHz processor
- 8 GB RAM

- We implemented prior techniques and our own using the LLVM compiler framework to create an abstract interpreter
- We used Apron for implementation of abstract domains and transfer functions
- And, to solve our constraints we used the mu-Z datalog engine inside Z3
- We evaluated our tool against prior work using 38 C programs
- These benchmarks were ones we wrote ourself as well as some from the Software Verification competition and linux device drivers
- All the tests were run on a 2.6 GHz processor with 8 GB RAM

Result Summary

- Prior work:

- Next, we'll look at the summarized results of our experiments
- First, applying prior work to our benchmarks resulted in 38 properties verified in 154 seconds
- Next, we tested our approach without checking if any of the interferences were infeasible using our constraint analysis: the only improvement we performed was delaying the join of environments across threads
- This resulted in 452 properties being verified in 421 seconds
- Next, we ran our approach delaying the join of interferences and also using the constraint analysis to remove infeasible interference combinations
- This resulted in 1,078 verified properties in 671 seconds
- Finally, we ran our technique with all the previously described optimizations
- This maintained the same 1,078 verified properties but dropped the runtime down to 480 seconds
- Overall, our technique with all the optimizations verified twenty eight times more properties with only an 3x increase in runtime

Result Summary

- Prior work:
  - 38 verified properties

- Next, we'll look at the summarized results of our experiments
- First, applying prior work to our benchmarks resulted in 38 properties verified in 154 seconds
- Next, we tested our approach without checking if any of the interferences were infeasible using our constraint analysis: the only improvement we performed was delaying the join of environments across threads
- This resulted in 452 properties being verified in 421 seconds
- Next, we ran our approach delaying the join of interferences and also using the constraint analysis to remove infeasible interference combinations
- This resulted in 1,078 verified properties in 671 seconds
- Finally, we ran our technique with all the previously described optimizations
- This maintained the same 1,078 verified properties but dropped the runtime down to 480 seconds
- Overall, our technique with all the optimizations verified twenty eight times more properties with only an 3x increase in runtime

Result Summary

- Prior work:
  - 38 verified properties
  - 154s runtime

- Next, we'll look at the summarized results of our experiments
- First, applying prior work to our benchmarks resulted in 38 properties verified in 154 seconds
- Next, we tested our approach without checking if any of the interferences were infeasible using our constraint analysis: the only improvement we performed was delaying the join of environments across threads
- This resulted in 452 properties being verified in 421 seconds
- Next, we ran our approach delaying the join of interferences and also using the constraint analysis to remove infeasible interference combinations
- This resulted in 1,078 verified properties in 671 seconds
- Finally, we ran our technique with all the previously described optimizations
- This maintained the same 1,078 verified properties but dropped the runtime down to 480 seconds
- Overall, our technique with all the optimizations verified twenty eight times more properties with only an 3x increase in runtime

Result Summary

- Prior work:
  - 38 verified properties
  - 154s runtime
- Without constraint pruning:

- Next, we'll look at the summarized results of our experiments
- First, applying prior work to our benchmarks resulted in 38 properties verified in 154 seconds
- Next, we tested our approach without checking if any of the interferences were infeasible using our constraint analysis: the only improvement we performed was delaying the join of environments across threads
- This resulted in 452 properties being verified in 421 seconds
- Next, we ran our approach delaying the join of interferences and also using the constraint analysis to remove infeasible interference combinations
- This resulted in 1,078 verified properties in 671 seconds
- Finally, we ran our technique with all the previously described optimizations
- This maintained the same 1,078 verified properties but dropped the runtime down to 480 seconds
- Overall, our technique with all the optimizations verified twenty eight times more properties with only an 3x increase in runtime

Result Summary

- Prior work:
  - 38 verified properties
  - 154s runtime
- Without constraint pruning:
  - 452 verified properties

- Next, we'll look at the summarized results of our experiments
- First, applying prior work to our benchmarks resulted in 38 properties verified in 154 seconds
- Next, we tested our approach without checking if any of the interferences were infeasible using our constraint analysis: the only improvement we performed was delaying the join of environments across threads
- This resulted in 452 properties being verified in 421 seconds
- Next, we ran our approach delaying the join of interferences and also using the constraint analysis to remove infeasible interference combinations
- This resulted in 1,078 verified properties in 671 seconds
- Finally, we ran our technique with all the previously described optimizations
- This maintained the same 1,078 verified properties but dropped the runtime down to 480 seconds
- Overall, our technique with all the optimizations verified twenty eight times more properties with only an 3x increase in runtime

Result Summary

- Prior work:
  - 38 verified properties
  - 154s runtime
- Without constraint pruning:
  - 452 verified properties
  - 421s runtime

- Next, we'll look at the summarized results of our experiments
- First, applying prior work to our benchmarks resulted in 38 properties verified in 154 seconds
- Next, we tested our approach without checking if any of the interferences were infeasible using our constraint analysis: the only improvement we performed was delaying the join of environments across threads
- This resulted in 452 properties being verified in 421 seconds
- Next, we ran our approach delaying the join of interferences and also using the constraint analysis to remove infeasible interference combinations
- This resulted in 1,078 verified properties in 671 seconds
- Finally, we ran our technique with all the previously described optimizations
- This maintained the same 1,078 verified properties but dropped the runtime down to 480 seconds
- Overall, our technique with all the optimizations verified twenty eight times more properties with only an 3x increase in runtime

Result Summary

- Prior work:
  - 38 verified properties
  - 154s runtime
- Without constraint pruning:
  - 452 verified properties
  - 421s runtime
- With constraints, no optimizations:

- Next, we'll look at the summarized results of our experiments
- First, applying prior work to our benchmarks resulted in 38 properties verified in 154 seconds
- Next, we tested our approach without checking if any of the interferences were infeasible using our constraint analysis: the only improvement we performed was delaying the join of environments across threads
- This resulted in 452 properties being verified in 421 seconds
- Next, we ran our approach delaying the join of interferences and also using the constraint analysis to remove infeasible interference combinations
- This resulted in 1,078 verified properties in 671 seconds
- Finally, we ran our technique with all the previously described optimizations
- This maintained the same 1,078 verified properties but dropped the runtime down to 480 seconds
- Overall, our technique with all the optimizations verified twenty eight times more properties with only an 3x increase in runtime

Result Summary

- Prior work:
  - 38 verified properties
  - 154s runtime
- Without constraint pruning:
  - 452 verified properties
  - 421s runtime
- With constraints, no optimizations:
  - 1,078 verified properties

- Next, we'll look at the summarized results of our experiments
- First, applying prior work to our benchmarks resulted in 38 properties verified in 154 seconds
- Next, we tested our approach without checking if any of the interferences were infeasible using our constraint analysis: the only improvement we performed was delaying the join of environments across threads
- This resulted in 452 properties being verified in 421 seconds
- Next, we ran our approach delaying the join of interferences and also using the constraint analysis to remove infeasible interference combinations
- This resulted in 1,078 verified properties in 671 seconds
- Finally, we ran our technique with all the previously described optimizations
- This maintained the same 1,078 verified properties but dropped the runtime down to 480 seconds
- Overall, our technique with all the optimizations verified twenty eight times more properties with only an 3x increase in runtime

Result Summary

- Prior work:
  - 38 verified properties
  - 154s runtime
- Without constraint pruning:
  - 452 verified properties
  - 421s runtime
- With constraints, no optimizations:
  - 1,078 verified properties
  - 671s runtime

- Next, we'll look at the summarized results of our experiments
- First, applying prior work to our benchmarks resulted in 38 properties verified in 154 seconds
- Next, we tested our approach without checking if any of the interferences were infeasible using our constraint analysis: the only improvement we performed was delaying the join of environments across threads
- This resulted in 452 properties being verified in 421 seconds
- Next, we ran our approach delaying the join of interferences and also using the constraint analysis to remove infeasible interference combinations
- This resulted in 1,078 verified properties in 671 seconds
- Finally, we ran our technique with all the previously described optimizations
- This maintained the same 1,078 verified properties but dropped the runtime down to 480 seconds
- Overall, our technique with all the optimizations verified twenty eight times more properties with only an 3x increase in runtime

Result Summary

- Prior work:
  - 38 verified properties
  - 154s runtime
- Without constraint pruning:
  - 452 verified properties
  - 421s runtime
- With constraints, no optimizations:
  - 1,078 verified properties
  - 671s runtime
- With optimizations:

- Next, we'll look at the summarized results of our experiments
- First, applying prior work to our benchmarks resulted in 38 properties verified in 154 seconds
- Next, we tested our approach without checking if any of the interferences were infeasible using our constraint analysis: the only improvement we performed was delaying the join of environments across threads
- This resulted in 452 properties being verified in 421 seconds
- Next, we ran our approach delaying the join of interferences and also using the constraint analysis to remove infeasible interference combinations
- This resulted in 1,078 verified properties in 671 seconds
- Finally, we ran our technique with all the previously described optimizations
- This maintained the same 1,078 verified properties but dropped the runtime down to 480 seconds
- Overall, our technique with all the optimizations verified twenty eight times more properties with only an 3x increase in runtime

Result Summary

- Prior work:
  - 38 verified properties
  - 154s runtime
- Without constraint pruning:
  - 452 verified properties
  - 421s runtime
- With constraints, no optimizations:
  - 1,078 verified properties
  - 671s runtime
- With optimizations:
  - 1,078 verified properties

- Next, we'll look at the summarized results of our experiments
- First, applying prior work to our benchmarks resulted in 38 properties verified in 154 seconds
- Next, we tested our approach without checking if any of the interferences were infeasible using our constraint analysis: the only improvement we performed was delaying the join of environments across threads
- This resulted in 452 properties being verified in 421 seconds
- Next, we ran our approach delaying the join of interferences and also using the constraint analysis to remove infeasible interference combinations
- This resulted in 1,078 verified properties in 671 seconds
- Finally, we ran our technique with all the previously described optimizations
- This maintained the same 1,078 verified properties but dropped the runtime down to 480 seconds
- Overall, our technique with all the optimizations verified twenty eight times more properties with only an 3x increase in runtime

Result Summary

- Prior work:
  - 38 verified properties
  - 154s runtime
- Without constraint pruning:
  - 452 verified properties
  - 421s runtime
- With constraints, no optimizations:
  - 1,078 verified properties
  - 671s runtime
- With optimizations:
  - 1,078 verified properties
  - 480s runtime

- Next, we'll look at the summarized results of our experiments
- First, applying prior work to our benchmarks resulted in 38 properties verified in 154 seconds
- Next, we tested our approach without checking if any of the interferences were infeasible using our constraint analysis: the only improvement we performed was delaying the join of environments across threads
- This resulted in 452 properties being verified in 421 seconds
- Next, we ran our approach delaying the join of interferences and also using the constraint analysis to remove infeasible interference combinations
- This resulted in 1,078 verified properties in 671 seconds
- Finally, we ran our technique with all the previously described optimizations
- This maintained the same 1,078 verified properties but dropped the runtime down to 480 seconds
- Overall, our technique with all the optimizations verified twenty eight times more properties with only an 3x increase in runtime

Result Summary

- Prior work:
  - 38 verified properties
  - 154s runtime
- Without constraint pruning:
  - 452 verified properties
  - 421s runtime
- With constraints, no optimizations:
  - 1,078 verified properties
  - 671s runtime
- With optimizations:
  - 1,078 verified properties
  - 480s runtime
- 28x more verified properties

- Next, we'll look at the summarized results of our experiments
- First, applying prior work to our benchmarks resulted in 38 properties verified in 154 seconds
- Next, we tested our approach without checking if any of the interferences were infeasible using our constraint analysis: the only improvement we performed was delaying the join of environments across threads
- This resulted in 452 properties being verified in 421 seconds
- Next, we ran our approach delaying the join of interferences and also using the constraint analysis to remove infeasible interference combinations
- This resulted in 1,078 verified properties in 671 seconds
- Finally, we ran our technique with all the previously described optimizations
- This maintained the same 1,078 verified properties but dropped the runtime down to 480 seconds
- Overall, our technique with all the optimizations verified twenty eight times more properties with only an 3x increase in runtime

Result Summary

- Prior work:
  - 38 verified properties
  - 154s runtime
- Without constraint pruning:
  - 452 verified properties
  - 421s runtime
- With constraints, no optimizations:
  - 1,078 verified properties
  - 671s runtime
- With optimizations:
  - 1,078 verified properties
  - 480s runtime
- 28x more verified properties
- 3x increase in time

- Next, we'll look at the summarized results of our experiments
- First, applying prior work to our benchmarks resulted in 38 properties verified in 154 seconds
- Next, we tested our approach without checking if any of the interferences were infeasible using our constraint analysis: the only improvement we performed was delaying the join of environments across threads
- This resulted in 452 properties being verified in 421 seconds
- Next, we ran our approach delaying the join of interferences and also using the constraint analysis to remove infeasible interference combinations
- This resulted in 1,078 verified properties in 671 seconds
- Finally, we ran our technique with all the previously described optimizations
- This maintained the same 1,078 verified properties but dropped the runtime down to 480 seconds
- Overall, our technique with all the optimizations verified twenty eight times more properties with only an 3x increase in runtime

Future Work

- Non-numerical domains

- We have many plans for future work for this project
- First, we'd like to see if we can use the same framework for non-numerical abstract interpretation based static analysis such as points-to, or shape analysis
- Second, during the previously described analysis we assumed the architecture was sequentially consistent
- Since this assumption does not actually hold for most processors, like ARM, and x86, we'd like to modify our constraints system and abstract interpreter in order to model these weak architectures
- And finally, the implementation of the algorithms we described were all sequential
- Since each thread is analyzed independently, the algorithm seems to be trivially paralleizable
- However, empirical evaluation of such parallization remains to be done

Future Work

- Non-numerical domains
- Non-sequentially consistent architectures

- We have many plans for future work for this project
- First, we'd like to see if we can use the same framework for non-numerical abstract interpretation based static analysis such as points-to, or shape analysis
- Second, during the previously described analysis we assumed the architecture was sequentially consistent
- Since this assumption does not actually hold for most processors, like ARM, and x86, we'd like to modify our constraints system and abstract interpreter in order to model these weak architectures
- And finally, the implementation of the algorithms we described were all sequential
- Since each thread is analyzed independently, the algorithm seems to be trivially paralleizable
- However, empirical evaluation of such parallization remains to be done

Future Work

- Non-numerical domains
- Non-sequentially consistent architectures
- Parallelization

- We have many plans for future work for this project
- First, we'd like to see if we can use the same framework for non-numerical abstract interpretation based static analysis such as points-to, or shape analysis
- Second, during the previously described analysis we assumed the architecture was sequentially consistent
- Since this assumption does not actually hold for most processors, like ARM, and x86, we'd like to modify our constraints system and abstract interpreter in order to model these weak architectures
- And finally, the implementation of the algorithms we described were all sequential
- Since each thread is analyzed independently, the algorithm seems to be trivially paralleizable
- However, empirical evaluation of such parallization remains to be done

Conclusion

- Flow-sensitive thread-modular abstract interpretation

- In conclusion, we presented a flow-sensitive thread-modular abstract interpretation technique
- We used two techniques in order to introduce flow-sensitivity to the analysis: first, we delayed the join of interferences across threads and second we removed infeasible inteference combinations using a system of lightweight constraints
- We evaluated our approach emperically on 38 C programs including linux device drivers
- Overall, our results showed that our technique can increase the number of verified properties 28 times compared to prior work
- This increase in the accuracy of the analysis came with only a 3x increase in runtime

Conclusion

- Flow-sensitive thread-modular abstract interpretation
- Delaying join of interferences

- In conclusion, we presented a flow-sensitive thread-modular abstract interpretation technique
- We used two techniques in order to introduce flow-sensitivity to the analysis: first, we delayed the join of interferences across threads and second we removed infeasible inteference combinations using a system of lightweight constraints
- We evaluated our approach emperically on 38 C programs including linux device drivers
- Overall, our results showed that our technique can increase the number of verified properties 28 times compared to prior work
- This increase in the accuracy of the analysis came with only a 3x increase in runtime

Conclusion

- Flow-sensitive thread-modular abstract interpretation
- Delaying join of interferences
- Removing infeasible interference combinations

- In conclusion, we presented a flow-sensitive thread-modular abstract interpretation technique
- We used two techniques in order to introduce flow-sensitivity to the analysis: first, we delayed the join of interferences across threads and second we removed infeasible inteference combinations using a system of lightweight constraints
- We evaluated our approach emperically on 38 C programs including linux device drivers
- Overall, our results showed that our technique can increase the number of verified properties 28 times compared to prior work
- This increase in the accuracy of the analysis came with only a 3x increase in runtime

Conclusion

- Flow-sensitive thread-modular abstract interpretation
- Delaying join of interferences
- Removing infeasible interference combinations
- Empirical analysis on C programs

- In conclusion, we presented a flow-sensitive thread-modular abstract interpretation technique
- We used two techniques in order to introduce flow-sensitivity to the analysis: first, we delayed the join of interferences across threads and second we removed infeasible inteference combinations using a system of lightweight constraints
- We evaluated our approach emperically on 38 C programs including linux device drivers
- Overall, our results showed that our technique can increase the number of verified properties 28 times compared to prior work
- This increase in the accuracy of the analysis came with only a 3x increase in runtime

Conclusion

- Flow-sensitive thread-modular abstract interpretation
- Delaying join of interferences
- Removing infeasible interference combinations
- Empirical analysis on C programs
- 28x increase in verified properties

- In conclusion, we presented a flow-sensitive thread-modular abstract interpretation technique
- We used two techniques in order to introduce flow-sensitivity to the analysis: first, we delayed the join of interferences across threads and second we removed infeasible inteference combinations using a system of lightweight constraints
- We evaluated our approach emperically on 38 C programs including linux device drivers
- Overall, our results showed that our technique can increase the number of verified properties 28 times compared to prior work
- This increase in the accuracy of the analysis came with only a 3x increase in runtime

Conclusion

- Flow-sensitive thread-modular abstract interpretation
- Delaying join of interferences
- Removing infeasible interference combinations
- Empirical analysis on C programs
- 28x increase in verified properties
- 3x increase in runtime

- In conclusion, we presented a flow-sensitive thread-modular abstract interpretation technique
- We used two techniques in order to introduce flow-sensitivity to the analysis: first, we delayed the join of interferences across threads and second we removed infeasible inteference combinations using a system of lightweight constraints
- We evaluated our approach emperically on 38 C programs including linux device drivers
- Overall, our results showed that our technique can increase the number of verified properties 28 times compared to prior work
- This increase in the accuracy of the analysis came with only a 3x increase in runtime

- *Dynamic Generation of Likely Invariants for Multithreaded Programs*, **Markus Kusano**, Arijit Chattopadhyay, Chao Wang. 37th International Conference on Software Engineering (ICSE). 2015.
- *Assertion Guided Abstraction: A Cooperative Optimization for Dynamic Partial Order Reduction*, **Markus Kusano** and Chao Wang. 29th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2015
- *CCmutator: A Mutation Generator for Concurrency Constructs in Multithreaded C/C++ Applications*, **Markus Kusano** and Chao Wang. 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2014.
- *Assertion Guided Symbolic Execution of Multithreaded Programs*, Shengjian Guo, **Markus Kusano**, Chao Wang, Zijiang Yang, Aarti Gupta. 23rd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE). 2015.
- *ConcBugAssist: Constraint Solving for Diagnosis and Repair of Concurrent Bugs*, Sepideh Khoshnood, **Markus Kusano**, Chao Wang. 2015 International Symposium on Software Testing and Analysis (ISSTA).

- This project is part of my prior work on testing, verification, and repair of concurrent programs

	Flow-insensitive		Flow-sensitive		F.-s. + Const.		F.-s. + Opt.	
Name	Tm. (s)	Verif.	Tm. (s)	Verif.	Tm. (s)	Verif.	Tm. (s)	Verif.
thread01	0.30	0	0.04	0	0.51	1	0.21	1
create01	0.01	0	0.02	0	0.13	1	0.15	1
create01	0.01	0	0.02	0	0.13	1	0.15	1
sync01	0.04	0	0.04	1	0.12	1	0.08	1
sync02	0.03	0	0.06	0	0.42	1	0.11	1
sync02	0.03	0	0.07	0	0.18	1	0.10	1
intra01	0.02	0	0.03	0	0.08	1	0.12	1
dekker1	0.09	0	10.5	0	20.0	1	3.86	1
fk2012	0.05	0	0.23	0	0.43	1	0.21	1
keybISR	0.04	0	0.16	0	0.46	2	0.16	2
ib700_01	0.08	0	0.08	0	0.09	1	0.12	1
ib700_02	1.25	0	0.88	0	0.96	1	1.19	1
ib700_03	10.8	0	17.6	40	14.5	81	14.6	81
i8xxtco_01	0.12	0	0.10	0	0.11	1	0.21	1
i8xxtco_02	1.69	0	0.89	0	1.42	1	1.29	1
i8xxtco_03	12.8	18	24.6	61	20.0	103	26.4	103
machz_01	0.18	0	0.14	0	0.16	1	0.26	1
machz_02	0.94	0	0.65	0	0.69	1	0.88	1
machz_03	7.96	0	41.3	42	68.2	83	35.6	83
mix_01	0.22	0	0.17	0	0.19	1	0.28	1
mix_02	4.75	1	7.17	31	4.96	62	6.52	62
pcwd_01	0.22	0	0.17	0	0.19	1	0.30	1
pcwd_02	32.4	0	59.9	40	26.5	81	32.9	81
sbc_01	0.51	0	0.65	0	1.35	1	0.55	1
sc1200_01	1.22	0	1.50	0	0.48	1	0.50	1
sc1200_02	13.1	0	62.8	62	243.8	93	186.3	93
smsc_01	0.25	0	0.21	0	0.28	1	0.39	1
smsc_02	3.33	0	18.2	1	21.9	24	7.26	24
sc520_01	0.51	0	1.15	0	0.59	1	0.64	1
sc520_02	37.5	0	128.8	39	53.7	81	43.0	81
wfwdt_01	0.49	0	1.24	0	0.58	1	0.64	1
wfwdt_02	77.7	0	260.4	0	98.3	101	85.7	101
wdt	1.48	0	1.04	0	0.86	1	1.67	1
wdt977_01	0.54	0	0.59	0	0.41	1	0.58	1
wdt977_02	24.1	0	49.5	32	97.03	93	106.2	93
wdt_pci	0.84	0	0.64	0	0.63	1	0.78	1
wdt_pci02	31.3	1	155.1	31	78.9	122	88.5	122
pcwdpci_01	46.3	18	107.0	72	76.7	128	54.9	128
Total	313.6	38	954.2	452	836.3	1079	703.5	1079