Dynamic Partial Order Reduction for Relaxed Memory Models

PLDI 2015

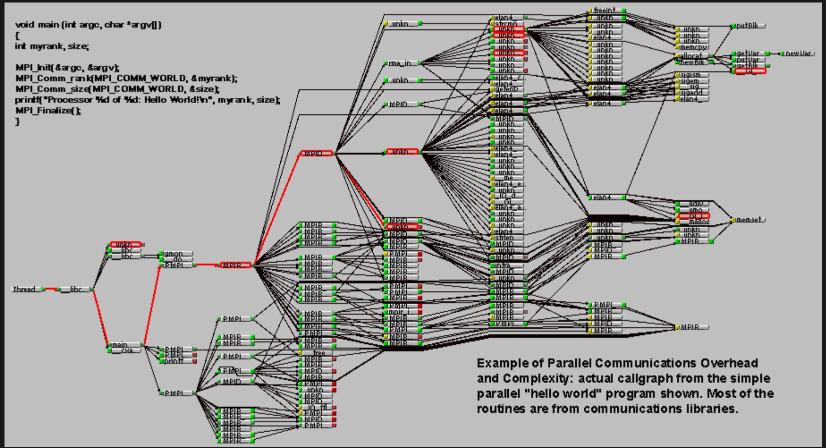
Naling Zhang, Markus Kusano, Chao Wang

June 16, 2015

Thanks for the introduction. My name is Markus and I'll be presenting this paper.

This is a collaborative work which obviously would not have been possible without the help from my colleagues Naling and Chao.

Introduction



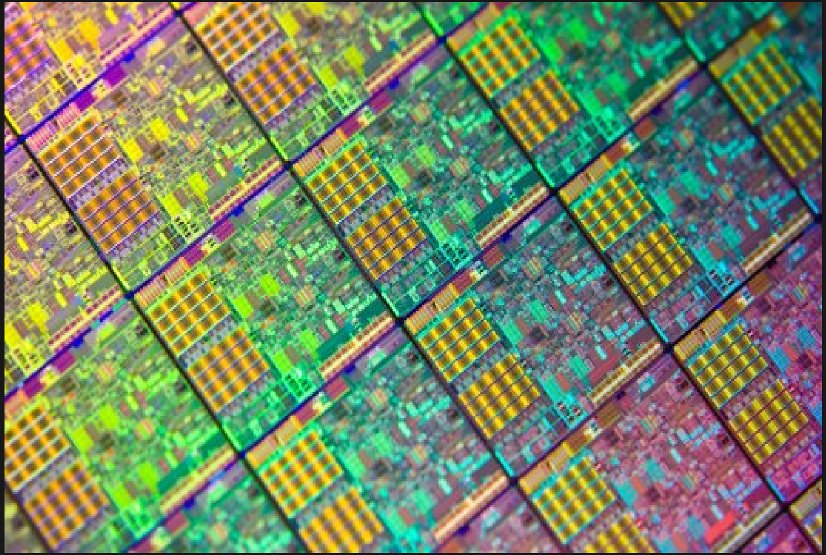
```
void main (int argc, char *argv[])
{
int myrank, size;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
printf("Processor %d of %d: Hello World!\n", myrank, size);
MPI_Finalize();
}
```

Example of Parallel Communications Overhead and Complexity: actual callgraph from the simple parallel "hello world" program shown. Most of the routines are from communications libraries.

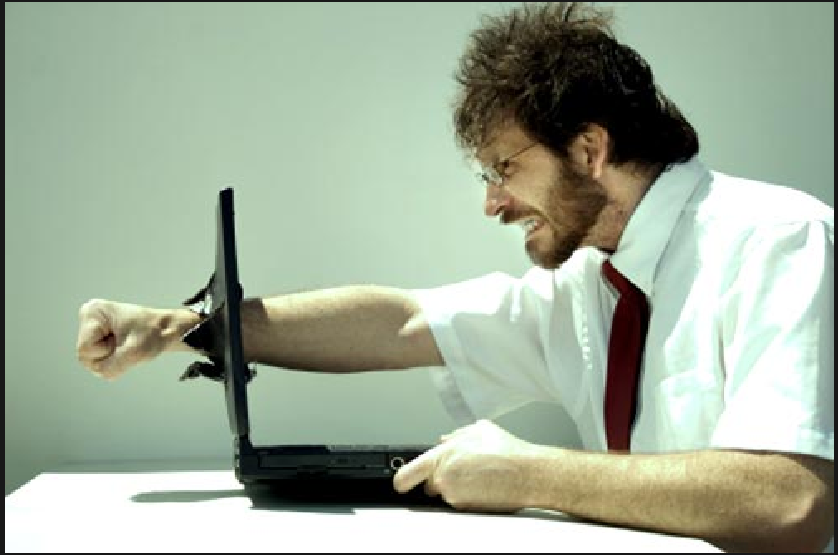
- We all can agree writing correct multithreaded programs is difficult
- The reason for this is the multitude of interactions between threads
- There are far too many combinations of orderings for a human to reason about
- For example, we can see here the call-graph for a seemingly simple program is a huge mess to understand.
- To make matters worse, to truely get high multi-threaded performance, the programmer needs to consider the relaxed memory architecture of their processor.
- As I'll show later, these relaxed hardware architectures can introduce non-intuitive behavior into the program
- This leaves programmers in a bad situation: writing high performance code for a relaxed memory model is difficult.
- As a result, it would be nice if we had automated testing and verification tools for multithreaded programs written under relaxed memory models

Introduction



- We all can agree writing correct multithreaded programs is difficult
- The reason for this is the multitude of interactions between threads
- There are far too many combinations of orderings for a human to reason about
- For example, we can see here the call-graph for a seemingly simple program is a huge mess to understand.
- To make matters worse, to truely get high multi-threaded performance, the programmer needs to consider the relaxed memory architecture of their processor.
- As I'll show later, these relaxed hardware architectures can introduce non-intuitive behavior into the program
- This leaves programmers in a bad situation: writing high performance code for a relaxed memory model is difficult.
- As a result, it would be nice if we had automated testing and verification tools for multithreaded programs written under relaxed memory models

Introduction



- We all can agree writing correct multithreaded programs is difficult
- The reason for this is the multitude of interactions between threads
- There are far too many combinations of orderings for a human to reason about
- For example, we can see here the call-graph for a seemingly simple program is a huge mess to understand.
- To make matters worse, to truely get high multi-threaded performance, the programmer needs to consider the relaxed memory architecture of their processor.
- As I'll show later, these relaxed hardware architectures can introduce non-intuitive behavior into the program
- This leaves programmers in a bad situation: writing high performance code for a relaxed memory model is difficult.
- As a result, it would be nice if we had automated testing and verification tools for multithreaded programs written under relaxed memory models

Introduction

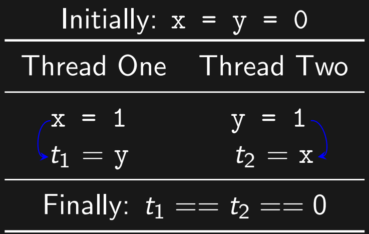
Initially: `x = y = 0`

Thread One	Thread Two
`x = 1`	`y = 1`
$t_1 = \texttt{y}$	$t_2 = \texttt{x}$

Finally: $t_1 == t_2 == 0$

- To continue, from the standpoint of the programmer, the weak-memory architecture can introduce subtle and non-intuitive behavior
- Consider this example, two threads are writing to shared memory locations x and y and then reading from y and x
- An allowable final state is that both t_1 and t_2 contain the value zero
- By simply considering all the interleavings of reads and writes from both processors, this final state should not be possible
- Slightly more formally, if we examine the ordering constraints required for this execution we can see that under typical sequentially consistent assumptions this execution is infeasible.
- First the order of statements within each thread should be consistent, the first statement within each thread happens before the second
- Additionally, in order for the final values to be zero, the last two statements of both threads must happen before the writes
- As we can see, the resulting constraints form a cycle so the resulting execution should be infeasible
- However, the observed final state is possible on common architectures such as Intel and AMD's TSO
- The reason for this behavior is that hardware designers permit the reordering of reads and writes within a thread to increase performance
- Because of this non-intuitive behavior, tools which automatically explore the relaxed-memory behavior of a program can help a designer to write correct programs

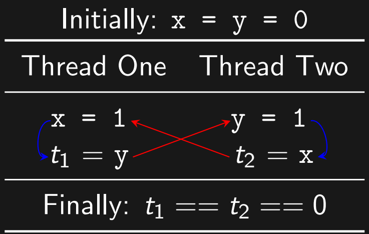
Introduction



Initially: x = y = 0	
Thread One	Thread Two
x = 1 $t_1 = \mathtt{y}$	y = 1 $t_2 = \mathtt{x}$
Finally: $t_1 == t_2 == 0$	

- To continue, from the standpoint of the programmer, the weak-memory architecture can introduce subtle and non-intuitive behavior
- Consider this example, two threads are writing to shared memory locations x and y and then reading from y and x
- An allowable final state is that both t_1 and t_2 contain the value zero
- By simply considering all the interleavings of reads and writes from both processors, this final state should not be possible
- Slightly more formally, if we examine the ordering constraints required for this execution we can see that under typical sequentially consistent assumptions this execution is infeasible.
- First the order of statements within each thread should be consistent, the first statement within each thread happens before the second
- Additionally, in order for the final values to be zero, the last two statements of both threads must happen before the writes
- As we can see, the resulting constraints form a cycle so the resulting execution should be infeasible
- However, the observed final state is possible on common architectures such as Intel and AMD's TSO
- The reason for this behavior is that hardware designers permit the reordering of reads and writes within a thread to increase performance
- Because of this non-intuitive behavior, tools which automatically explore the relaxed-memory behavior of a program can help a designer to write correct programs

Introduction



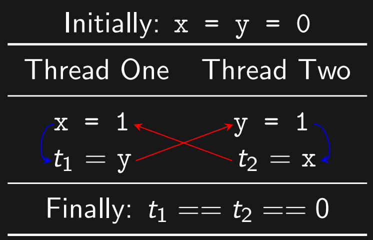
Initially: `x = y = 0`

Thread One	Thread Two
`x = 1`	`y = 1`
$t_1 = \mathtt{y}$	$t_2 = \mathtt{x}$

Finally: $t_1 == t_2 == 0$

- To continue, from the standpoint of the programmer, the weak-memory architecture can introduce subtle and non-intuitive behavior
- Consider this example, two threads are writing to shared memory locations `x` and `y` and then reading from `y` and `x`
- An allowable final state is that both t_1 and t_2 contain the value zero
- By simply considering all the interleavings of reads and writes from both processors, this final state should not be possible
- Slightly more formally, if we examine the ordering constraints required for this execution we can see that under typical sequentially consistent assumptions this execution is infeasible.
- First the order of statements within each thread should be consistent, the first statement within each thread happens before the second
- Additionally, in order for the final values to be zero, the last two statements of both threads must happen before the writes
- As we can see, the resulting constraints form a cycle so the resulting execution should be infeasible
- However, the observed final state is possible on common architectures such as Intel and AMD's TSO
- The reason for this behavior is that hardware designers permit the reordering of reads and writes within a thread to increase performance
- Because of this non-intuitive behavior, tools which automatically explore the relaxed-memory behavior of a program can help a designer to write correct programs

Introduction



Initially: `x = y = 0`

Thread One	Thread Two
`x = 1`	`y = 1`
$t_1 =$ `y`	$t_2 =$ `x`

Finally: $t_1 == t_2 == 0$

Sequentially *inconsistent* behavior is allowed on modern architectures.

- To continue, from the standpoint of the programmer, the weak-memory architecture can introduce subtle and non-intuitive behavior
- Consider this example, two threads are writing to shared memory locations `x` and `y` and then reading from `y` and `x`
- An allowable final state is that both t_1 and t_2 contain the value zero
- By simply considering all the interleavings of reads and writes from both processors, this final state should not be possible
- Slightly more formally, if we examine the ordering constraints required for this execution we can see that under typical sequentially consistent assumptions this execution is infeasible.
- First the order of statements within each thread should be consistent, the first statement within each thread happens before the second
- Additionally, in order for the final values to be zero, the last two statements of both threads must happen before the writes
- As we can see, the resulting constraints form a cycle so the resulting execution should be infeasible
- However, the observed final state is possible on common architectures such as Intel and AMD's TSO
- The reason for this behavior is that hardware designers permit the reordering of reads and writes within a thread to increase performance
- Because of this non-intuitive behavior, tools which automatically explore the relaxed-memory behavior of a program can help a designer to write correct programs

Introduction

- Over 10x increase in number of runs

- However, the additional complexity introduced by weak memory affects not only the programmer but also the verification tool
- The increase in allowable behavior in the program means the verifier needs to explore a much larger state space
- In esscense, the verifier needs to not only test all the sequentially consistent behavior but also all the weak memory behavior
- In our experiments, we found that the number of permissible runs moving from a sequentially consistent architecture to a relaxed architecture increases by over 10 times
- This explosion in the number of permissible runs decreases the efficiency of the verifier
- As such, we require intelligent pruning methods to identify redundancies across executions
- More specifically, often the weak memory behavior can be shown to be equivalent to the sequentially consistent behavior
- As a result, a naive exploration leads to redundancies and far too much runtime overhead

Introduction

- Over 10x increase in number of runs

- However, the additional complexity introduced by weak memory affects not only the programmer but also the verification tool
- The increase in allowable behavior in the program means the verifier needs to explore a much larger state space
- In esscense, the verifier needs to not only test all the sequentially consistent behavior but also all the weak memory behavior
- In our experiments, we found that the number of permissible runs moving from a sequentially consistent architecture to a relaxed architecture increases by over 10 times
- This explosion in the number of permissible runs decreases the efficiency of the verifier
- As such, we require intelligent pruning methods to identify redundancies across executions
- More specifically, often the weak memory behavior can be shown to be equivalent to the sequentially consistent behavior
- As a result, a naive exploration leads to redundancies and far too much runtime overhead

Contribution

- DPOR for TSO & PSO
- Automatically detect weak-memory bugs
- Unified consideration of sequentially consistent and relaxed behaviors
- Significant reduction in runtime overhead

- Our contribution aims to solve the previous problems
- We created a unified framework to test concurrent programs under weak-memory hardware architectures
- The two architectures we consider are TSO and PSO
- TSO is the architecture used by Intel and AMD while PSO is used by SPARC
- Specifically, we extend the well known DPOR algorithm to handle weak memory models
- The end result of our work is the detection of concurrency bugs including those caused by the relaxed memory architecture
- Our framework is unified in that it considers all executions, be they sequentially consistent or relaxed, in the same fashion
- As a result, we can detect rundundancies and greatly reduce the runtime overhead

Overview

Introduction

Background

- Relaxed Memory Models
- Stateless Model Checking

rInspect

Experiments

Conclusion

Next, I will provide some more detailed background on relaxed memory models. For this talk, I will focus only on TSO but the extension to PSO, as shown in our paper, is fairly straightforward.

Relaxed Memory Models

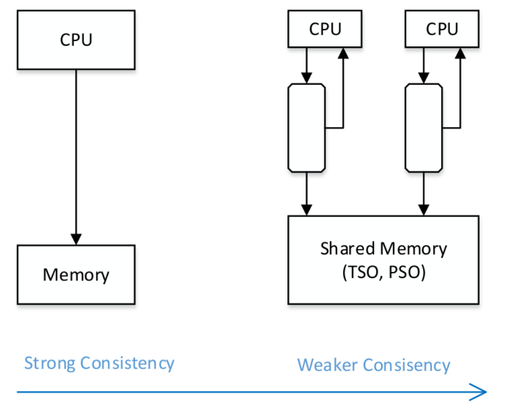
- TSO: x86
- PSO: SPARC

- In our work, we focus on two relaxed memory models: TSO and PSO
- TSO is used on Intel and AMD processors while PSO is used by SPARC
- Both models allow for two relaxations of the ordering of statements both within and across threads.
- The first relaxation is the use of store buffers
- At a high level, store buffers allow for writes to memory to be delayed for an arbitrary ammount of time
- The second is the ordering of statements within a thread
- In essense, the memory models allow for certain reads and writes to be ordered differently than the programmers direct specification
- The next few slides will go over these points in detail

Relaxed Memory Models

- TSO: x86
- PSO: SPARC

1. Store buffers
2. Intra-thread program order



- In our work, we focus on two relaxed memory models: TSO and PSO
- TSO is used on Intel and AMD processors while PSO is used by SPARC
- Both models allow for two relaxations of the ordering of statements both within and across threads.
- The first relaxation is the use of store buffers
- At a high level, store buffers allow for writes to memory to be delayed for an arbitrary ammount of time
- The second is the ordering of statements within a thread
- In essense, the memory models allow for certain reads and writes to be ordered differently than the programmers direct specification
- The next few slides will go over these points in detail

Store Buffers

Initially: `x = y = 0`

Thread One	Thread Two
`x = 1`	`y = 1`
$t_1 = \texttt{y}$	$t_2 = \texttt{x}$

Finally: $t_1 == t_2 == 0$

- At a high level, a store buffer is a FIFO queue for a given hardware thread containing pending memory writes
- Operationally, a thread will always read its most recent pending value in its own store buffer
- However, the store buffers are thread local: the values in the buffer are invisible to other threads
- In essense, this allows for writes in one thread to be arbitrarily delayed
- If we look back at the previous example we can see that the behavior is a consequence of store buffering
- For example, here is one possible ordering allowing the behavior
- First, thread one executes its memory write but the result is buffered
- Next, thread two executes its write to y which is also buffered
- After that, thread one reads the value of y. Since thread two's write is in the buffer, it is invisible to thread one. So, thread one reads the value of zero
- Similarly, thread two reads the value of zero when it reads x
- Finally, this results in the two reads both reading zero
- Ontop of this, the final quirk with stores buffers is that at thread always reads its buffered value
- In other words, if thread one were to read `x` at any point after its initial write it would always read the value of one even if the write was still buffered.

Store Buffers

Initially: `x = y = 0`	
Thread One	Thread Two
`x = 1`	`y = 1`
$t_1 = \texttt{y}$	$t_2 = \texttt{x}$
Finally: $t_1 == t_2 == 0$	

Thread One	Thread Two	Thread One's Buffer	Thread Two's Buffer
`x = 1`		`x = 1`	∅

- At a high level, a store buffer is a FIFO queue for a given hardware thread containing pending memory writes
- Operationally, a thread will always read its most recent pending value in its own store buffer
- However, the store buffers are thread local: the values in the buffer are invisible to other threads
- In essense, this allows for writes in one thread to be arbitrarily delayed
- If we look back at the previous example we can see that the behavior is a consequence of store buffering
- For example, here is one possible ordering allowing the behavior
- First, thread one executes its memory write but the result is buffered
- Next, thread two executes its write to y which is also buffered
- After that, thread one reads the value of y. Since thread two's write is in the buffer, it is invisible to thread one. So, thread one reads the value of zero
- Similarly, thread two reads the value of zero when it reads x
- Finally, this results in the two reads both reading zero
- Ontop of this, the final quirk with stores buffers is that at thread always reads its buffered value
- In other words, if thread one were to read `x` at any point after its initial write it would always read the value of one even if the write was still buffered.

Store Buffers

Initially: `x = y = 0`

Thread One	Thread Two
`x = 1`	`y = 1`
$t_1 = \mathtt{y}$	$t_2 = \mathtt{x}$

Finally: $t_1 == t_2 == 0$

Thread One	Thread Two	Thread One's Buffer	Thread Two's Buffer
`x = 1`		`x = 1`	∅
	`y = 1`	`x = 1`	`y = 1`

- At a high level, a store buffer is a FIFO queue for a given hardware thread containing pending memory writes
- Operationally, a thread will always read its most recent pending value in its own store buffer
- However, the store buffers are thread local: the values in the buffer are invisible to other threads
- In essense, this allows for writes in one thread to be arbitrarily delayed
- If we look back at the previous example we can see that the behavior is a consequence of store buffering
- For example, here is one possible ordering allowing the behavior
- First, thread one executes its memory write but the result is buffered
- Next, thread two executes its write to y which is also buffered
- After that, thread one reads the value of y. Since thread two's write is in the buffer, it is invisible to thread one. So, thread one reads the value of zero
- Similarly, thread two reads the value of zero when it reads x
- Finally, this results in the two reads both reading zero
- Ontop of this, the final quirk with stores buffers is that at thread always reads its buffered value
- In other words, if thread one were to read `x` at any point after its initial write it would always read the value of one even if the write was still buffered.

Store Buffers

Initially: x = y = 0

Thread One	Thread Two
x = 1	y = 1
$t_1 = y$	$t_2 = x$

Finally: $t_1 == t_2 == 0$

Thread One	Thread Two	Thread One's Buffer	Thread Two's Buffer
x = 1		x = 1	∅
	y = 1	x = 1	y = 1
t_1 = y == 0		x = 1	y = 1

- At a high level, a store buffer is a FIFO queue for a given hardware thread containing pending memory writes
- Operationally, a thread will always read its most recent pending value in its own store buffer
- However, the store buffers are thread local: the values in the buffer are invisible to other threads
- In essense, this allows for writes in one thread to be arbitrarily delayed
- If we look back at the previous example we can see that the behavior is a consequence of store buffering
- For example, here is one possible ordering allowing the behavior
- First, thread one executes its memory write but the result is buffered
- Next, thread two executes its write to y which is also buffered
- After that, thread one reads the value of y. Since thread two's write is in the buffer, it is invisible to thread one. So, thread one reads the value of zero
- Similarly, thread two reads the value of zero when it reads x
- Finally, this results in the two reads both reading zero
- Ontop of this, the final quirk with stores buffers is that at thread always reads its buffered value
- In other words, if thread one were to read x at any point after its initial write it would always read the value of one even if the write was still buffered.

Store Buffers

Initially: x = y = 0

Thread One	Thread Two
x = 1	y = 1
$t_1 = \mathrm{y}$	$t_2 = \mathrm{x}$

Finally: $t_1 == t_2 == 0$

Thread One	Thread Two	Thread One's Buffer	Thread Two's Buffer
x = 1		x = 1	∅
	y = 1	x = 1	y = 1
t_1 = y == 0		x = 1	y = 1
	t_2 = x == 0	x = 1	y = 1

- At a high level, a store buffer is a FIFO queue for a given hardware thread containing pending memory writes
- Operationally, a thread will always read its most recent pending value in its own store buffer
- However, the store buffers are thread local: the values in the buffer are invisible to other threads
- In essense, this allows for writes in one thread to be arbitrarily delayed
- If we look back at the previous example we can see that the behavior is a consequence of store buffering
- For example, here is one possible ordering allowing the behavior
- First, thread one executes its memory write but the result is buffered
- Next, thread two executes its write to y which is also buffered
- After that, thread one reads the value of y. Since thread two's write is in the buffer, it is invisible to thread one. So, thread one reads the value of zero
- Similarly, thread two reads the value of zero when it reads x
- Finally, this results in the two reads both reading zero
- Ontop of this, the final quirk with stores buffers is that at thread always reads its buffered value
- In other words, if thread one were to read x at any point after its initial write it would always read the value of one even if the write was still buffered.

Intra-thread Program Order

Thread One	Thread Two
x = 1	y = 1
$t_1 = \mathrm{y}$	$t_2 = \mathrm{x}$

- In addition to delays across threads from store buffers, relaxed memory architectures also allow statements within a thread to be reorded.
- Under TSO, a write and a following read to different memory locations can be re-ordered.
- Consider the same program from before
- If we examine the write and read in thread one we can see that based on the TSO rules they can be reorded
- Conceptually, this means the program can also execute as if it were written with these statements reodered
- Similarly, the two statements within thread two may also be reorded resulting in a different potential program
- And, just for completation, it is also possible for both re-orderings to take place resulting in a fourth potential program
- These subtle reorderings can cause bugs in software mutual exclusion algorithms such as Dekker's Lock as well as in design patterns like double-checked locking

Intra-thread Program Order

Thread One	Thread Two
x = 1	y = 1
$t_1 = y$	$t_2 = x$

- In addition to delays across threads from store buffers, relaxed memory architectures also allow statements within a thread to be reorded.
- Under TSO, a write and a following read to different memory locations can be re-ordered.
- Consider the same program from before
- If we examine the write and read in thread one we can see that based on the TSO rules they can be reorded
- Conceptually, this means the program can also execute as if it were written with these statements reodered
- Similarly, the two statements within thread two may also be reorded resulting in a different potential program
- And, just for completation, it is also possible for both re-orderings to take place resulting in a fourth potential program
- These subtle reorderings can cause bugs in software mutual exclusion algorithms such as Dekker's Lock as well as in design patterns like double-checked locking

Intra-thread Program Order

Thread One	Thread Two
x = 1	y = 1
$t_1 = $ y	$t_2 = $ x

Thread One	Thread Two
t_1 = y	y = 1
x = 1	$t_2 = $ x

- In addition to delays across threads from store buffers, relaxed memory architectures also allow statements within a thread to be reorded.
- Under TSO, a write and a following read to different memory locations can be re-ordered.
- Consider the same program from before
- If we examine the write and read in thread one we can see that based on the TSO rules they can be reorded
- Conceptually, this means the program can also execute as if it were written with these statements reodered
- Similarly, the two statements within thread two may also be reorded resulting in a different potential program
- And, just for completation, it is also possible for both re-orderings to take place resulting in a fourth potential program
- These subtle reorderings can cause bugs in software mutual exclusion algorithms such as Dekker's Lock as well as in design patterns like double-checked locking

Intra-thread Program Order

Thread One	Thread Two
x = 1	y = 1
$t_1 = y$	$t_2 = x$

Thread One	Thread Two
$t_1 = y$	y = 1
x = 1	$t_2 = x$

- In addition to delays across threads from store buffers, relaxed memory architectures also allow statements within a thread to be reorded.
- Under TSO, a write and a following read to different memory locations can be re-ordered.
- Consider the same program from before
- If we examine the write and read in thread one we can see that based on the TSO rules they can be reorded
- Conceptually, this means the program can also execute as if it were written with these statements reodered
- Similarly, the two statements within thread two may also be reorded resulting in a different potential program
- And, just for completation, it is also possible for both re-orderings to take place resulting in a fourth potential program
- These subtle reorderings can cause bugs in software mutual exclusion algorithms such as Dekker's Lock as well as in design patterns like double-checked locking

Intra-thread Program Order

Thread One	Thread Two
x = 1	y = 1
$t_1 = $ y	$t_2 = $ x

Thread One	Thread Two
$t_1 = $ y	y = 1
x = 1	$t_2 = $ x

Thread One	Thread Two
x = 1	$t_2 = $ x
$t_1 = $ y	y = 1

- In addition to delays across threads from store buffers, relaxed memory architectures also allow statements within a thread to be reorded.
- Under TSO, a write and a following read to different memory locations can be re-ordered.
- Consider the same program from before
- If we examine the write and read in thread one we can see that based on the TSO rules they can be reorded
- Conceptually, this means the program can also execute as if it were written with these statements reodered
- Similarly, the two statements within thread two may also be reorded resulting in a different potential program
- And, just for completation, it is also possible for both re-orderings to take place resulting in a fourth potential program
- These subtle reorderings can cause bugs in software mutual exclusion algorithms such as Dekker's Lock as well as in design patterns like double-checked locking

Intra-thread Program Order

Thread One	Thread Two
x = 1	y = 1
$t_1 = y$	$t_2 = x$

Thread One	Thread Two
$t_1 = y$	y = 1
x = 1	$t_2 = x$

Thread One	Thread Two
x = 1	$t_2 = x$
$t_1 = y$	y = 1

Thread One	Thread Two
$t_1 = y$	$t_2 = x$
x = 1	y = 1

- In addition to delays across threads from store buffers, relaxed memory architectures also allow statements within a thread to be reorded.
- Under TSO, a write and a following read to different memory locations can be re-ordered.
- Consider the same program from before
- If we examine the write and read in thread one we can see that based on the TSO rules they can be reorded
- Conceptually, this means the program can also execute as if it were written with these statements reodered
- Similarly, the two statements within thread two may also be reorded resulting in a different potential program
- And, just for completation, it is also possible for both re-orderings to take place resulting in a fourth potential program
- These subtle reorderings can cause bugs in software mutual exclusion algorithms such as Dekker's Lock as well as in design patterns like double-checked locking

RMM Summary

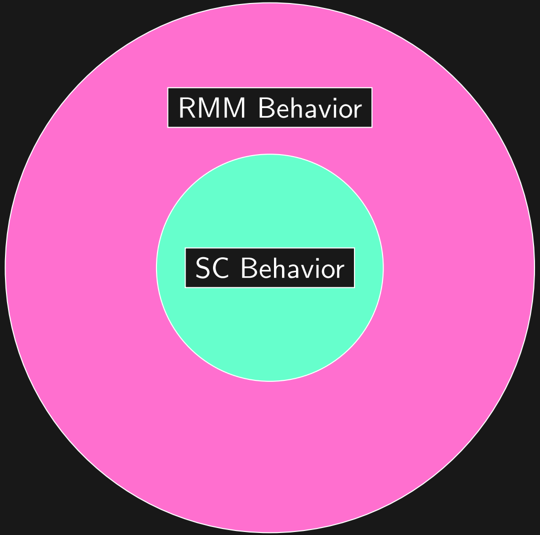
- Just to summarize, we can look at a very high-level at the consequences of relaxed memory models
- First, we can see the space of all possible behaviors of a program assuming a sequentially consistent memory model
- This is the realm focused on by many prior works
- However, when considering relaxed memory models the behavior space increases dramatically
- In general, the relaxed memory model still contains all the sequentially consistent behavior but extends it with additional behavior.
- In the case of TSO and PSO, this additional behavior comes from store-buffering and intra-thread relaxations
- The increase in behavior not only makes the programmer confused but also increases the burdern on automated testers and verifiers
- As I'll show in the remainder of this talk, our goal is to mitiagte the additional complextity to make the verifier more efficent

RMM Summary

SC Behavior

- Just to summarize, we can look at a very high-level at the consequences of relaxed memory models
- First, we can see the space of all possible behaviors of a program assuming a sequentially consistent memory model
- This is the realm focused on by many prior works
- However, when considering relaxed memory models the behavior space increases dramatically
- In general, the relaxed memory model still contains all the sequentially consistent behavior but extends it with additional behavior.
- In the case of TSO and PSO, this additional behavior comes from store-buffering and intra-thread relaxations
- The increase in behavior not only makes the programmer confused but also increases the burdern on automated testers and verifiers
- As I'll show in the remainder of this talk, our goal is to mitiagte the additional complextity to make the verifier more efficent

RMM Summary



- Just to summarize, we can look at a very high-level at the consequences of relaxed memory models
- First, we can see the space of all possible behaviors of a program assuming a sequentially consistent memory model
- This is the realm focused on by many prior works
- However, when considering relaxed memory models the behavior space increases dramatically
- In general, the relaxed memory model still contains all the sequentially consistent behavior but extends it with additional behavior.
- In the case of TSO and PSO, this additional behavior comes from store-buffering and intra-thread relaxations
- The increase in behavior not only makes the programmer confused but also increases the burdern on automated testers and verifiers
- As I'll show in the remainder of this talk, our goal is to mitiagte the additional complextity to make the verifier more efficent

Overview

Introduction

Background

- Relaxed Memory Models
- Stateless Model Checking

rInspect

Experiments

Conclusion

Next, I will provide a brief introduction to stateless model checking with dynamic partial order reduction.

Stateless Model Checking + Dynamic Partial Order Reduction

- Dynamic analysis
- Explore all *relevant* traces
- Sound [Flanagan and Godefroid, POPL 2005]

- Stateless model checking is a dynamic analysis
- The goal of the analysis is to explore all relevant traces of the program
- This is similar to a traditional model checker which explores all reachable states except that states are abstracted to traces
- The key word here is relevant traces: using dynamic partial order reduction we can explore a subset of all traces of the program
- Even though a subset is explored, the algorithm guarantees to explore all behavior relevant to safety properties and deadlocks
- In other words, the analysis itself is sound. It dynamically performs the verification of a multithreaded program for a given program input

Dynamic Partial Order Reduction (DPOR)

Thread One	Thread Two
`x = 1`	`x = 20`
`a = 1`	`b = 20`

- Although algorithmically DPOR is somewhat complex, I believe the operation of the algorithm is somewhat intuitive
- So, instead of going into the low-level details I'll introduce the algorithm with an example
- Consider this program where two threads both access a shared variable `x` and then individually modify variables `a` and `b`.
- Since there are two instructions in each thread, naively enumerating all possible schedules leads to 4 different possibilities.
- However, as DPOR can automatically deduce, two of these executions are redundant
- Initially, DPOR executes the program with an arbitrary schedule to generate an initial trace
- Examining this trace, we can find two statements which DPOR considers as dependent
- At a high level, dependent statements are those who's order could affect the results of the program
- Since the two modifications of `x` by both threads are modifying the same variable they are considered as dependent
- Then, DPOR re-executes the program such that thread two executes before thread one
- Examining this trace, we can again see the two statements involving `x` are seemingly dependent
- However, since we already explored the trace where thread one executes first we do not need to repeat it
- At this point, we have tested all the behavior of the program after only two executions compared to the naive exploration of four

Dynamic Partial Order Reduction (DPOR)

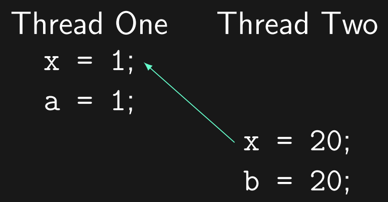
```
Thread One    Thread Two
  x = 1         x = 20
  a = 1         b = 20
```

```
Thread One    Thread Two
  x = 1;
  a = 1;
                x = 20;
                b = 20;
```

- Although algorithmically DPOR is somewhat complex, I believe the operation of the algorithm is somewhat intuitive
- So, instead of going into the low-level details I'll introduce the algorithm with an example
- Consider this program where two threads both access a shared variable `x` and then individually modify variables `a` and `b`.
- Since there are two instructions in each thread, naively enumerating all possible schedules leads to 4 different possibilities.
- However, as DPOR can automatically deduce, two of these executions are redundant
- Initially, DPOR executes the program with an arbitrary schedule to generate an initial trace
- Examining this trace, we can find two statements which DPOR considers as dependent
- At a high level, dependent statements are those who's order could affect the results of the program
- Since the two modifications of `x` by both threads are modifying the same variable they are considered as dependent
- Then, DPOR re-executes the program such that thread two executes before thread one
- Examining this trace, we can again see the two statements involving `x` are seemingly dependent
- However, since we already explored the trace where thread one executes first we do not need to repeat it
- At this point, we have tested all the behavior of the program after only two executions compared to the naive exploration of four

Dynamic Partial Order Reduction (DPOR)

Thread One	Thread Two
`x = 1`	`x = 20`
`a = 1`	`b = 20`

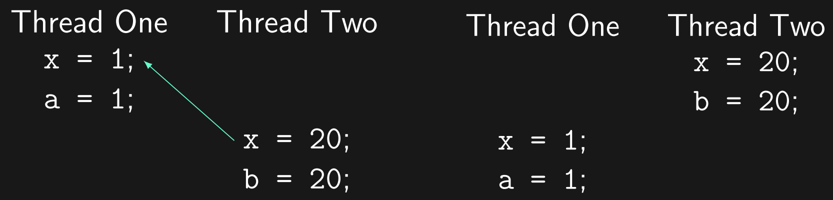


Thread One	Thread Two
`x = 1;`	
`a = 1;`	
	`x = 20;`
	`b = 20;`

- Although algorithmically DPOR is somewhat complex, I believe the operation of the algorithm is somewhat intuitive
- So, instead of going into the low-level details I'll introduce the algorithm with an example
- Consider this program where two threads both access a shared variable `x` and then individually modify variables `a` and `b`.
- Since there are two instructions in each thread, naively enumerating all possible schedules leads to 4 different possibilities.
- However, as DPOR can automatically deduce, two of these executions are redundant
- Initially, DPOR executes the program with an arbitrary schedule to generate an initial trace
- Examining this trace, we can find two statements which DPOR considers as dependent
- At a high level, dependent statements are those who's order could affect the results of the program
- Since the two modifications of `x` by both threads are modifying the same variable they are considered as dependent
- Then, DPOR re-executes the program such that thread two executes before thread one
- Examining this trace, we can again see the two statements involving `x` are seemingly dependent
- However, since we already explored the trace where thread one executes first we do not need to repeat it
- At this point, we have tested all the behavior of the program after only two executions compared to the naive exploration of four

Dynamic Partial Order Reduction (DPOR)

Thread One	Thread Two
x = 1	x = 20
a = 1	b = 20

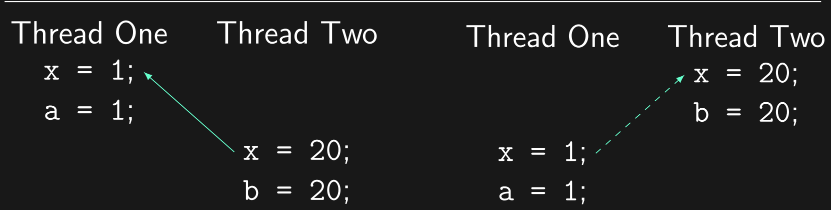


```
Thread One     Thread Two          Thread One     Thread Two
  x = 1;                                            x = 20;
  a = 1;                                            b = 20;
                 x = 20;             x = 1;
                 b = 20;             a = 1;
```

- Although algorithmically DPOR is somewhat complex, I believe the operation of the algorithm is somewhat intuitive
- So, instead of going into the low-level details I'll introduce the algorithm with an example
- Consider this program where two threads both access a shared variable `x` and then individually modify variables `a` and `b`.
- Since there are two instructions in each thread, naively enumerating all possible schedules leads to 4 different possibilities.
- However, as DPOR can automatically deduce, two of these executions are redundant
- Initially, DPOR executes the program with an arbitrary schedule to generate an initial trace
- Examining this trace, we can find two statements which DPOR considers as dependent
- At a high level, dependent statements are those who's order could affect the results of the program
- Since the two modifications of `x` by both threads are modifying the same variable they are considered as dependent
- Then, DPOR re-executes the program such that thread two executes before thread one
- Examining this trace, we can again see the two statements involving `x` are seemingly dependent
- However, since we already explored the trace where thread one executes first we do not need to repeat it
- At this point, we have tested all the behavior of the program after only two executions compared to the naive exploration of four

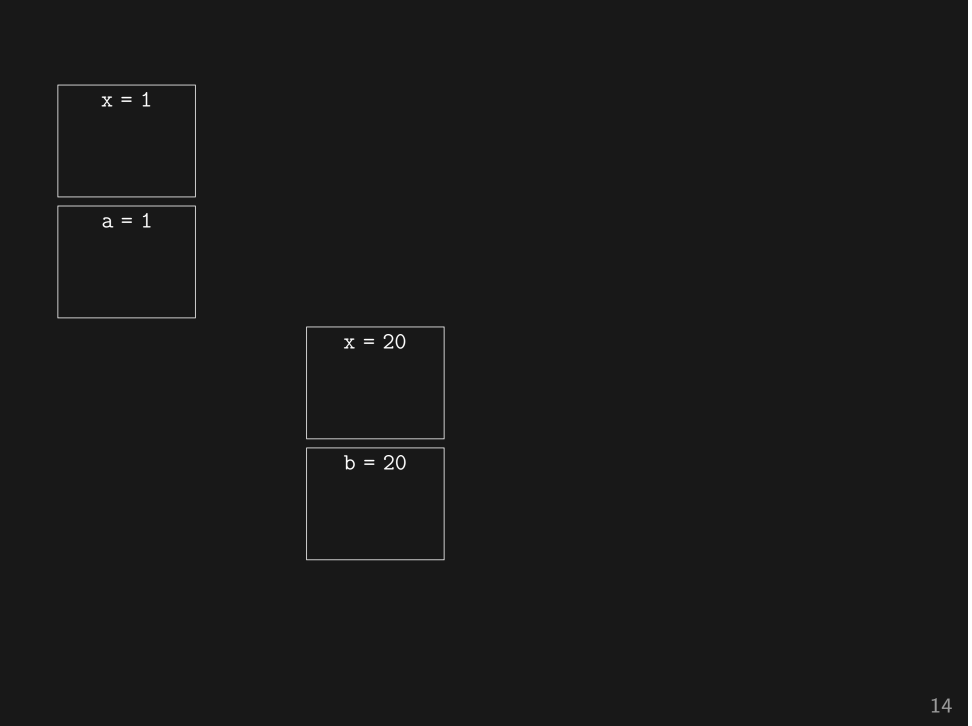
Dynamic Partial Order Reduction (DPOR)

```
Thread One    Thread Two
  x = 1         x = 20
  a = 1         b = 20
```



```
Thread One      Thread Two        Thread One      Thread Two
  x = 1;                                            x = 20;
  a = 1;                                            b = 20;
                  x = 20;           x = 1;
                  b = 20;           a = 1;
```

- Although algorithmically DPOR is somewhat complex, I believe the operation of the algorithm is somewhat intuitive
- So, instead of going into the low-level details I'll introduce the algorithm with an example
- Consider this program where two threads both access a shared variable `x` and then individually modify variables `a` and `b`.
- Since there are two instructions in each thread, naively enumerating all possible schedules leads to 4 different possibilities.
- However, as DPOR can automatically deduce, two of these executions are redundant
- Initially, DPOR executes the program with an arbitrary schedule to generate an initial trace
- Examining this trace, we can find two statements which DPOR considers as dependent
- At a high level, dependent statements are those who's order could affect the results of the program
- Since the two modifications of `x` by both threads are modifying the same variable they are considered as dependent
- Then, DPOR re-executes the program such that thread two executes before thread one
- Examining this trace, we can again see the two statements involving `x` are seemingly dependent
- However, since we already explored the trace where thread one executes first we do not need to repeat it
- At this point, we have tested all the behavior of the program after only two executions compared to the naive exploration of four



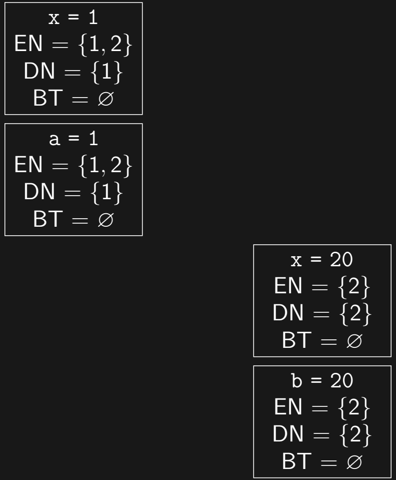
x = 1

a = 1

x = 20

b = 20

- Going into slightly more detail, we can look at the bookkeeping data structures used by the algorithm
- Here is the first trace executed in the previous example; thread one executes first followed by thread two
- Each statement in the program contains three sets: the enabled set, the done set, and the backtrack set
- The items inside of each set are thread IDs
- At the first statement, either thread can execute so both thread one and thread two are enabled
- Since we've already executed thread one at this point, the done set contains thread one
- And finally, the backtrack set, which contains statements we need to execute, is currently empty
- Next, when we identify the two dependent accesses to x across threads we update the backtrack set to explore the alternate execution on the next run
- On the subsequent execution, we switch the order of the accesses to x
- Again, the algorithm finds the two accesses to x as dependent. However, since thread one is already in the done set of the first statement, we do not need to test this schedule again.
- Finally, since there are no more remaining backtrack points to test, the analysis finishes after two runs



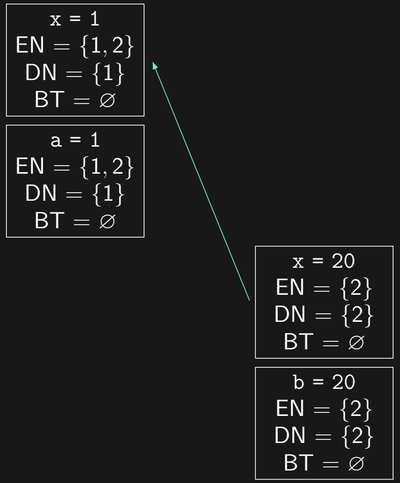
`x = 1`
EN $= \{1, 2\}$
DN $= \{1\}$
BT $= \varnothing$

`a = 1`
EN $= \{1, 2\}$
DN $= \{1\}$
BT $= \varnothing$

`x = 20`
EN $= \{2\}$
DN $= \{2\}$
BT $= \varnothing$

`b = 20`
EN $= \{2\}$
DN $= \{2\}$
BT $= \varnothing$

- Going into slightly more detail, we can look at the bookkeeping data structures used by the algorithm
- Here is the first trace executed in the previous example; thread one executes first followed by thread two
- Each statement in the program contains three sets: the enabled set, the done set, and the backtrack set
- The items inside of each set are thread IDs
- At the first statement, either thread can execute so both thread one and thread two are enabled
- Since we've already executed thread one at this point, the done set contains thread one
- And finally, the backtrack set, which contains statements we need to execute, is currently empty
- Next, when we identify the two dependent accesses to x across threads we update the backtrack set to explore the alternate execution on the next run
- On the subsequent execution, we switch the order of the accesses to x
- Again, the algorithm finds the two accesses to x as dependent. However, since thread one is already in the done set of the first statement, we do not need to test this schedule again.
- Finally, since there are no more remaining backtrack points to test, the analysis finishes after two runs



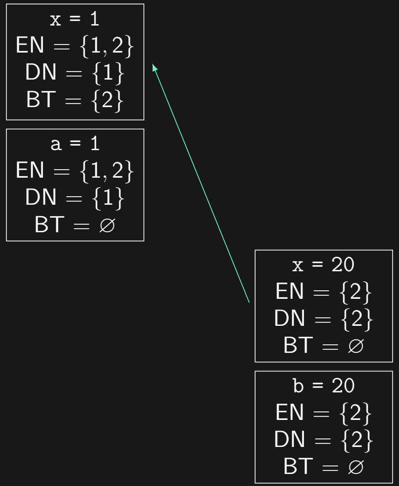
x = 1
$EN = \{1, 2\}$
$DN = \{1\}$
$BT = \varnothing$

a = 1
$EN = \{1, 2\}$
$DN = \{1\}$
$BT = \varnothing$

x = 20
$EN = \{2\}$
$DN = \{2\}$
$BT = \varnothing$

b = 20
$EN = \{2\}$
$DN = \{2\}$
$BT = \varnothing$

- Going into slightly more detail, we can look at the bookkeeping data structures used by the algorithm
- Here is the first trace executed in the previous example; thread one executes first followed by thread two
- Each statement in the program contains three sets: the enabled set, the done set, and the backtrack set
- The items inside of each set are thread IDs
- At the first statement, either thread can execute so both thread one and thread two are enabled
- Since we've already executed thread one at this point, the done set contains thread one
- And finally, the backtrack set, which contains statements we need to execute, is currently empty
- Next, when we identify the two dependent accesses to x across threads we update the backtrack set to explore the alternate execution on the next run
- On the subsequent execution, we switch the order of the accesses to x
- Again, the algorithm finds the two accesses to x as dependent. However, since thread one is already in the done set of the first statement, we do not need to test this schedule again.
- Finally, since there are no more remaining backtrack points to test, the analysis finishes after two runs



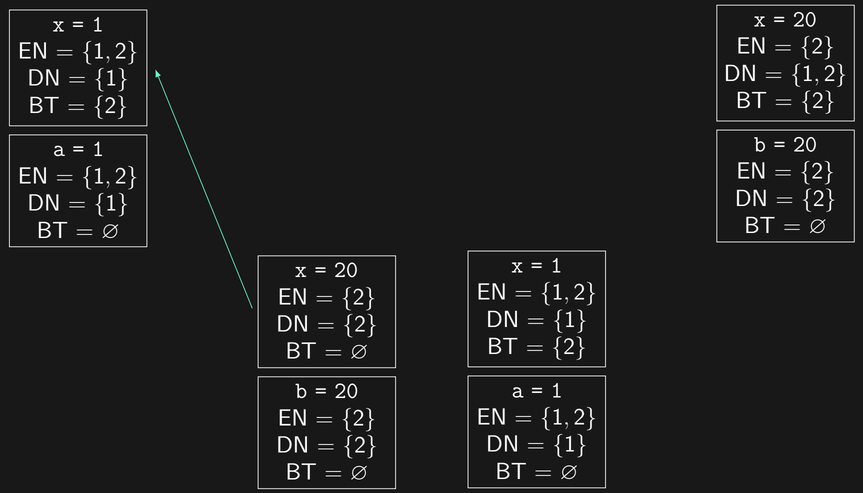
`x = 1`
$\mathsf{EN} = \{1, 2\}$
$\mathsf{DN} = \{1\}$
$\mathsf{BT} = \{2\}$

`a = 1`
$\mathsf{EN} = \{1, 2\}$
$\mathsf{DN} = \{1\}$
$\mathsf{BT} = \varnothing$

`x = 20`
$\mathsf{EN} = \{2\}$
$\mathsf{DN} = \{2\}$
$\mathsf{BT} = \varnothing$

`b = 20`
$\mathsf{EN} = \{2\}$
$\mathsf{DN} = \{2\}$
$\mathsf{BT} = \varnothing$

- Going into slightly more detail, we can look at the bookkeeping data structures used by the algorithm
- Here is the first trace executed in the previous example; thread one executes first followed by thread two
- Each statement in the program contains three sets: the enabled set, the done set, and the backtrack set
- The items inside of each set are thread IDs
- At the first statement, either thread can execute so both thread one and thread two are enabled
- Since we've already executed thread one at this point, the done set contains thread one
- And finally, the backtrack set, which contains statements we need to execute, is currently empty
- Next, when we identify the two dependent accesses to x across threads we update the backtrack set to explore the alternate execution on the next run
- On the subsequent execution, we switch the order of the accesses to x
- Again, the algorithm finds the two accesses to x as dependent. However, since thread one is already in the done set of the first statement, we do not need to test this schedule again.
- Finally, since there are no more remaining backtrack points to test, the analysis finishes after two runs



x = 1
$\text{EN} = \{1, 2\}$
$\text{DN} = \{1\}$
$\text{BT} = \{2\}$

a = 1
$\text{EN} = \{1, 2\}$
$\text{DN} = \{1\}$
$\text{BT} = \varnothing$

x = 20
$\text{EN} = \{2\}$
$\text{DN} = \{2\}$
$\text{BT} = \varnothing$

b = 20
$\text{EN} = \{2\}$
$\text{DN} = \{2\}$
$\text{BT} = \varnothing$

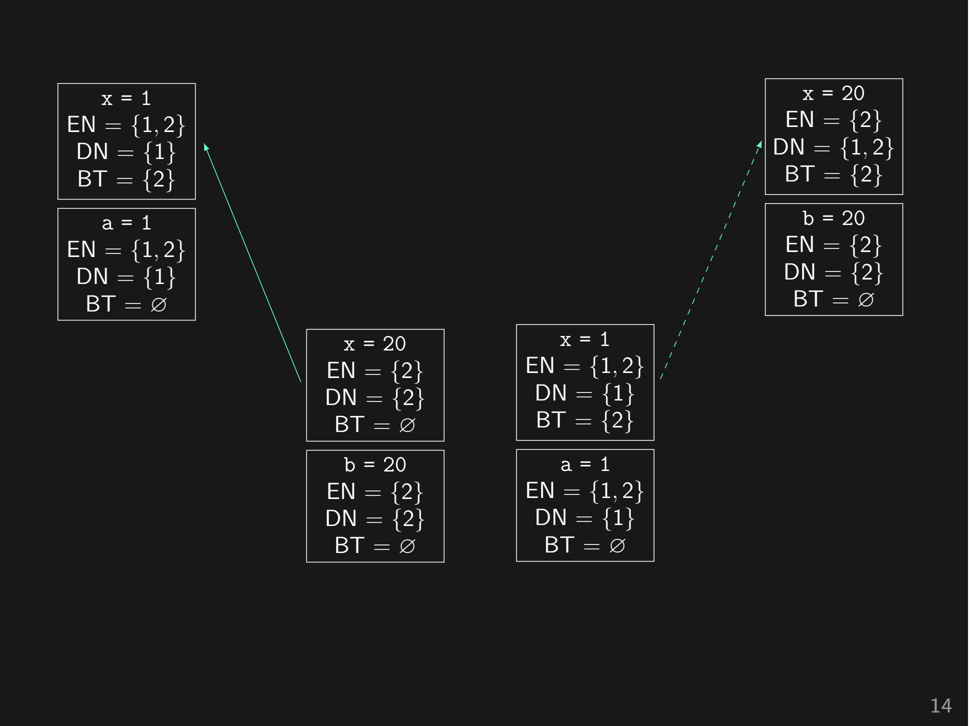
x = 1
$\text{EN} = \{1, 2\}$
$\text{DN} = \{1\}$
$\text{BT} = \{2\}$

a = 1
$\text{EN} = \{1, 2\}$
$\text{DN} = \{1\}$
$\text{BT} = \varnothing$

x = 20
$\text{EN} = \{2\}$
$\text{DN} = \{1, 2\}$
$\text{BT} = \{2\}$

b = 20
$\text{EN} = \{2\}$
$\text{DN} = \{2\}$
$\text{BT} = \varnothing$

- Going into slightly more detail, we can look at the bookkeeping data structures used by the algorithm
- Here is the first trace executed in the previous example; thread one executes first followed by thread two
- Each statement in the program contains three sets: the enabled set, the done set, and the backtrack set
- The items inside of each set are thread IDs
- At the first statement, either thread can execute so both thread one and thread two are enabled
- Since we've already executed thread one at this point, the done set contains thread one
- And finally, the backtrack set, which contains statements we need to execute, is currently empty
- Next, when we identify the two dependent accesses to x across threads we update the backtrack set to explore the alternate execution on the next run
- On the subsequent execution, we switch the order of the accesses to x
- Again, the algorithm finds the two accesses to x as dependent. However, since thread one is already in the done set of the first statement, we do not need to test this schedule again.
- Finally, since there are no more remaining backtrack points to test, the analysis finishes after two runs



x = 1
$\mathsf{EN} = \{1, 2\}$
$\mathsf{DN} = \{1\}$
$\mathsf{BT} = \{2\}$

a = 1
$\mathsf{EN} = \{1, 2\}$
$\mathsf{DN} = \{1\}$
$\mathsf{BT} = \varnothing$

x = 20
$\mathsf{EN} = \{2\}$
$\mathsf{DN} = \{2\}$
$\mathsf{BT} = \varnothing$

b = 20
$\mathsf{EN} = \{2\}$
$\mathsf{DN} = \{2\}$
$\mathsf{BT} = \varnothing$

x = 1
$\mathsf{EN} = \{1, 2\}$
$\mathsf{DN} = \{1\}$
$\mathsf{BT} = \{2\}$

a = 1
$\mathsf{EN} = \{1, 2\}$
$\mathsf{DN} = \{1\}$
$\mathsf{BT} = \varnothing$

x = 20
$\mathsf{EN} = \{2\}$
$\mathsf{DN} = \{1, 2\}$
$\mathsf{BT} = \{2\}$

b = 20
$\mathsf{EN} = \{2\}$
$\mathsf{DN} = \{2\}$
$\mathsf{BT} = \varnothing$

- Going into slightly more detail, we can look at the bookkeeping data structures used by the algorithm
- Here is the first trace executed in the previous example; thread one executes first followed by thread two
- Each statement in the program contains three sets: the enabled set, the done set, and the backtrack set
- The items inside of each set are thread IDs
- At the first statement, either thread can execute so both thread one and thread two are enabled
- Since we've already executed thread one at this point, the done set contains thread one
- And finally, the backtrack set, which contains statements we need to execute, is currently empty
- Next, when we identify the two dependent accesses to x across threads we update the backtrack set to explore the alternate execution on the next run
- On the subsequent execution, we switch the order of the accesses to x
- Again, the algorithm finds the two accesses to x as dependent. However, since thread one is already in the done set of the first statement, we do not need to test this schedule again.
- Finally, since there are no more remaining backtrack points to test, the analysis finishes after two runs

Overview

Introduction

Background

Relaxed Memory Models

Stateless Model Checking

rInspect

Experiments

Conclusion

Now that we've covered relaxed memory models and stateless model checking, I'll introduce our new tool, rInspect.

rInspect: Overview & Related Work

- Generalization of DPOR algorithm [Flanagan and Godefroid, POPL 2005]
- Non-persistent-set based techniques [Abdulla et al., TACAS 2015]
- Static program transformation [Alglave et al., ESOP 2013]
- SAT based [Alglave et al., CAV 2013]
- Stateful model checking [Linden and Wolper, TACAS 2013]

- At a high level, our implementation is a generalization of DPOR for relaxed memory models
- We slightly modify the fundamental DPOR data structures and relations such that the sequentially consistent and relaxed behaviors are considered simultaneously
- In this way, we gain all the benefits of prior work on partial-order reduction and fit within the ecosystem of prior work on the analysis of weak-memory programs
- First, our approach handles both inter- and intra-thread relaxations using persistent sets
- Our technique is also purely dynamic: similar to the original DPOR algorithm everything is based on the runtime behavior of the program allowing for very accurate results
- This differs from approaches to model weak-memory behaviors using program transformation: over-approximated static results cause false-positives
- We also do not use any SAT or SMT solvers so we avoid the runtime scalability issues associated with these approaches
- Finally, our approach is stateless: it does not store any of the values stored in memory avoiding memory explosion issues

Modeling Store Buffers

- $W_\tau()$: store buffer dequeue
- Enabled Set: $\{\ \langle \text{Thread ID}, \text{Statements} \rangle\ \}$

- During the dynamic analysis, we model store buffers as described previously
- To do so, we introduce a special function called W-sub-tau.
- This function dequeues the thread's store buffer
- To match the store-buffer semantics, we allow any thread to dequeue its store buffer at any point during the execution of the program
- However, the fundamental DPOR data-structures do not easily map to this concept
- So, we modified them slightly: instead of being a set of thread IDs, we expanded them to be a set of maps from thread IDs to statements.
- This may be somewhat confusing right now but I will show an example of how this works in practice

Modeling Store Buffers

`buf(x = 1)`

$$\mathsf{EN} = \left\{ \begin{array}{l} (1, \{W_\tau(), t1 = y\}), \\ (2, \{y = 1\}) \end{array} \right\}$$

- Here is the same store buffer example we've been looking at
- We will examine one trace of its execution to show how we model store buffers
- For simplicity, we will just look at the enabled sets; the done and backtrack sets are operationally similar to before
- The writes to memory by the two threads are replaced, instead, by a thread-local buffered write
- Examining the first statement, we can see that thread one can either flush its buffer or execute its next statement
- Thread two can only execute its next statement since its buffer is empty
- At the next statement, thread one can only flush its buffer since it executed its final statement
- When thread two executes, its enabled set is analogous to thread one's: after its first statement it can either flush its buffer or execute its next statement. For the sake of space, I hid the actual contents of the enabled sets.
- Finally, when thread one executes its final statement, the flush of its buffer, we can see there is a dependent access between thread twos read and the buffer flush
- As a result, we update the backtrack set such that the buffer flush occurs before the read of x
- While we are not going into the algorithmic details, notice how this process of identifying and flipping dependent statements is essentially the same as in the original DPOR algorithm.
- In this way, we gain all the pruning power of DPOR while considering a relaxed memory model

Modeling Store Buffers

`buf(x = 1)`

$$\mathsf{EN} = \left\{ \begin{matrix} (1, \{W_\tau(), t1 = y\}), \\ (2, \{y = 1\}) \end{matrix} \right\}$$

`t₁ = y`

$$\mathsf{EN} = \left\{ \begin{matrix} (1, \{W_\tau()\}), \\ (2, \{y = 1\}) \end{matrix} \right\}$$

- Here is the same store buffer example we've been looking at
- We will examine one trace of its execution to show how we model store buffers
- For simplicity, we will just look at the enabled sets; the done and backtrack sets are operationally similar to before
- The writes to memory by the two threads are replaced, instead, by a thread-local buffered write
- Examining the first statement, we can see that thread one can either flush its buffer or execute its next statement
- Thread two can only execute its next statement since its buffer is empty
- At the next statement, thread one can only flush its buffer since it executed its final statement
- When thread two executes, its enabled set is analogous to thread one's: after its first statement it can either flush its buffer or execute its next statement. For the sake of space, I hid the actual contents of the enabled sets.
- Finally, when thread one executes its final statement, the flush of its buffer, we can see there is a dependent access between thread twos read and the buffer flush
- As a result, we update the backtrack set such that the buffer flush occurs before the read of x
- While we are not going into the algorithmic details, notice how this process of identifying and flipping dependent statements is essentially the same as in the original DPOR algorithm.
- In this way, we gain all the pruning power of DPOR while considering a relaxed memory model

Modeling Store Buffers

`buf(x = 1)`

$$\mathsf{EN} = \left\{ \begin{array}{l} (1, \{W_\tau(), t1 = y\}), \\ (2, \{y = 1\}) \end{array} \right\}$$

t_1 `= y`

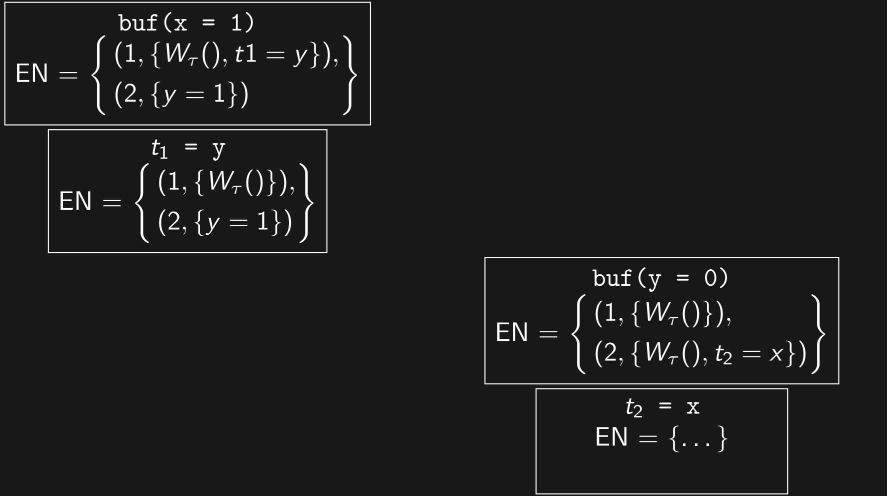
$$\mathsf{EN} = \left\{ \begin{array}{l} (1, \{W_\tau()\}), \\ (2, \{y = 1\}) \end{array} \right\}$$

`buf(y = 0)`

$$\mathsf{EN} = \left\{ \begin{array}{l} (1, \{W_\tau()\}), \\ (2, \{W_\tau(), t_2 = x\}) \end{array} \right\}$$

- Here is the same store buffer example we've been looking at
- We will examine one trace of its execution to show how we model store buffers
- For simplicity, we will just look at the enabled sets; the done and backtrack sets are operationally similar to before
- The writes to memory by the two threads are replaced, instead, by a thread-local buffered write
- Examining the first statement, we can see that thread one can either flush its buffer or execute its next statement
- Thread two can only execute its next statement since its buffer is empty
- At the next statement, thread one can only flush its buffer since it executed its final statement
- When thread two executes, its enabled set is analogous to thread one's: after its first statement it can either flush its buffer or execute its next statement. For the sake of space, I hid the actual contents of the enabled sets.
- Finally, when thread one executes its final statement, the flush of its buffer, we can see there is a dependent access between thread twos read and the buffer flush
- As a result, we update the backtrack set such that the buffer flush occurs before the read of x
- While we are not going into the algorithmic details, notice how this process of identifying and flipping dependent statements is essentially the same as in the original DPOR algorithm.
- In this way, we gain all the pruning power of DPOR while considering a relaxed memory model

Modeling Store Buffers



`buf(x = 1)`

$$\mathsf{EN} = \left\{ \begin{matrix} (1, \{W_\tau(), t1 = y\}), \\ (2, \{y = 1\}) \end{matrix} \right\}$$

`t1 = y`

$$\mathsf{EN} = \left\{ \begin{matrix} (1, \{W_\tau()\}), \\ (2, \{y = 1\}) \end{matrix} \right\}$$

`buf(y = 0)`

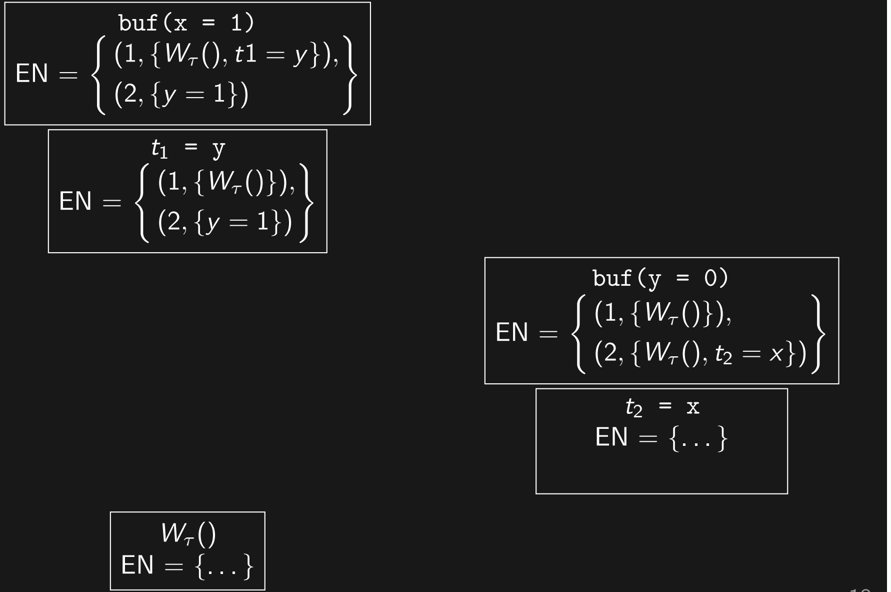
$$\mathsf{EN} = \left\{ \begin{matrix} (1, \{W_\tau()\}), \\ (2, \{W_\tau(), t_2 = x\}) \end{matrix} \right\}$$

`t2 = x`

$$\mathsf{EN} = \{\dots\}$$

- Here is the same store buffer example we've been looking at
- We will examine one trace of its execution to show how we model store buffers
- For simplicity, we will just look at the enabled sets; the done and backtrack sets are operationally similar to before
- The writes to memory by the two threads are replaced, instead, by a thread-local buffered write
- Examining the first statement, we can see that thread one can either flush its buffer or execute its next statement
- Thread two can only execute its next statement since its buffer is empty
- At the next statement, thread one can only flush its buffer since it executed its final statement
- When thread two executes, its enabled set is analogous to thread one's: after its first statement it can either flush its buffer or execute its next statement. For the sake of space, I hid the actual contents of the enabled sets.
- Finally, when thread one executes its final statement, the flush of its buffer, we can see there is a dependent access between thread twos read and the buffer flush
- As a result, we update the backtrack set such that the buffer flush occurs before the read of x
- While we are not going into the algorithmic details, notice how this process of identifying and flipping dependent statements is essentially the same as in the original DPOR algorithm.
- In this way, we gain all the pruning power of DPOR while considering a relaxed memory model

Modeling Store Buffers



`buf(x = 1)`

$$\mathsf{EN} = \left\{ \begin{array}{l} (1, \{W_\tau(), t1 = y\}), \\ (2, \{y = 1\}) \end{array} \right\}$$

t_1 `= y`

$$\mathsf{EN} = \left\{ \begin{array}{l} (1, \{W_\tau()\}), \\ (2, \{y = 1\}) \end{array} \right\}$$

`buf(y = 0)`

$$\mathsf{EN} = \left\{ \begin{array}{l} (1, \{W_\tau()\}), \\ (2, \{W_\tau(), t_2 = x\}) \end{array} \right\}$$

t_2 `= x`

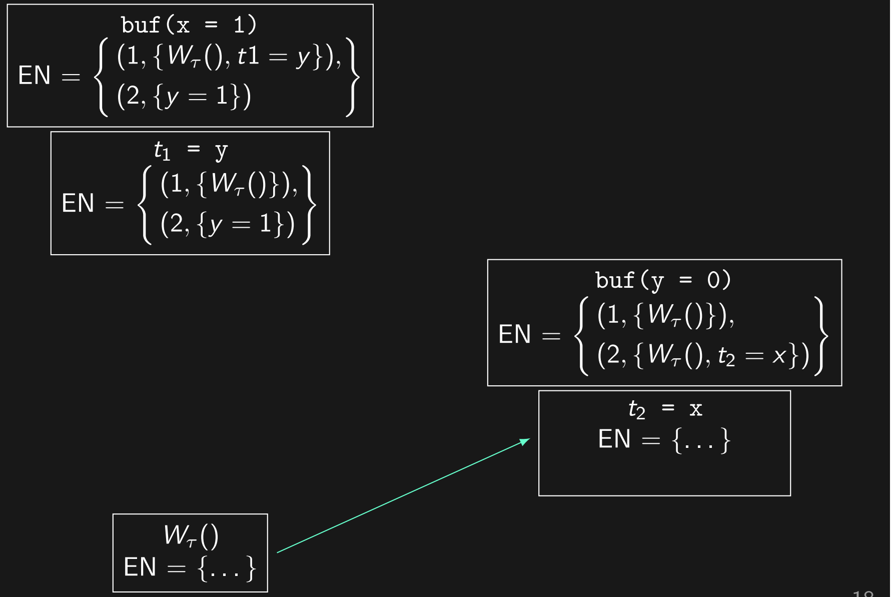
$$\mathsf{EN} = \{\dots\}$$

$W_\tau()$

$$\mathsf{EN} = \{\dots\}$$

- Here is the same store buffer example we've been looking at
- We will examine one trace of its execution to show how we model store buffers
- For simplicity, we will just look at the enabled sets; the done and backtrack sets are operationally similar to before
- The writes to memory by the two threads are replaced, instead, by a thread-local buffered write
- Examining the first statement, we can see that thread one can either flush its buffer or execute its next statement
- Thread two can only execute its next statement since its buffer is empty
- At the next statement, thread one can only flush its buffer since it executed its final statement
- When thread two executes, its enabled set is analogous to thread one's: after its first statement it can either flush its buffer or execute its next statement. For the sake of space, I hid the actual contents of the enabled sets.
- Finally, when thread one executes its final statement, the flush of its buffer, we can see there is a dependent access between thread twos read and the buffer flush
- As a result, we update the backtrack set such that the buffer flush occurs before the read of x
- While we are not going into the algorithmic details, notice how this process of identifying and flipping dependent statements is essentially the same as in the original DPOR algorithm.
- In this way, we gain all the pruning power of DPOR while considering a relaxed memory model

Modeling Store Buffers



`buf(x = 1)`

$$\mathsf{EN} = \left\{ \begin{array}{l} (1, \{W_\tau(), t1 = y\}), \\ (2, \{y = 1\}) \end{array} \right\}$$

t_1 `= y`

$$\mathsf{EN} = \left\{ \begin{array}{l} (1, \{W_\tau()\}), \\ (2, \{y = 1\}) \end{array} \right\}$$

`buf(y = 0)`

$$\mathsf{EN} = \left\{ \begin{array}{l} (1, \{W_\tau()\}), \\ (2, \{W_\tau(), t_2 = x\}) \end{array} \right\}$$

t_2 `= x`

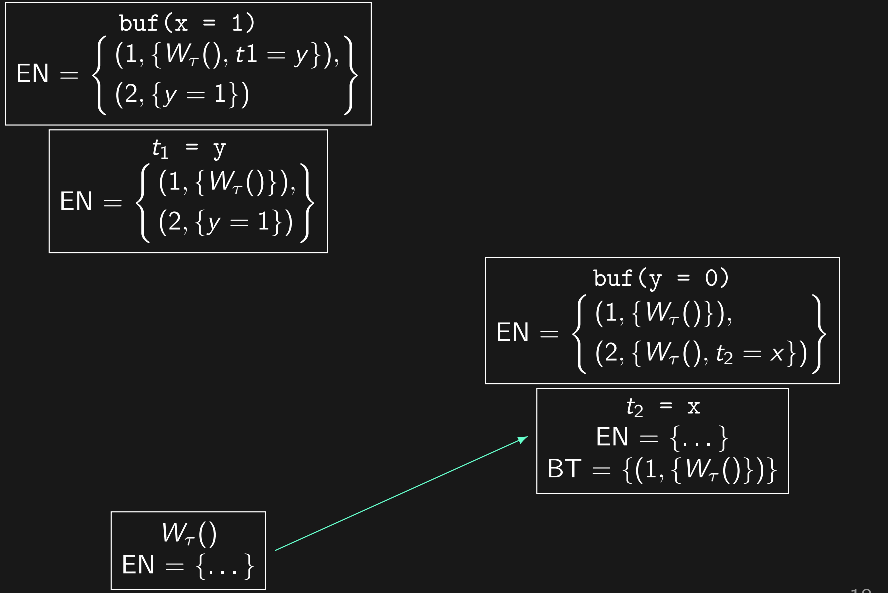
$$\mathsf{EN} = \{\dots\}$$

$W_\tau()$

$$\mathsf{EN} = \{\dots\}$$

- Here is the same store buffer example we've been looking at
- We will examine one trace of its execution to show how we model store buffers
- For simplicity, we will just look at the enabled sets; the done and backtrack sets are operationally similar to before
- The writes to memory by the two threads are replaced, instead, by a thread-local buffered write
- Examining the first statement, we can see that thread one can either flush its buffer or execute its next statement
- Thread two can only execute its next statement since its buffer is empty
- At the next statement, thread one can only flush its buffer since it executed its final statement
- When thread two executes, its enabled set is analogous to thread one's: after its first statement it can either flush its buffer or execute its next statement. For the sake of space, I hid the actual contents of the enabled sets.
- Finally, when thread one executes its final statement, the flush of its buffer, we can see there is a dependent access between thread twos read and the buffer flush
- As a result, we update the backtrack set such that the buffer flush occurs before the read of x
- While we are not going into the algorithmic details, notice how this process of identifying and flipping dependent statements is essentially the same as in the original DPOR algorithm.
- In this way, we gain all the pruning power of DPOR while considering a relaxed memory model

Modeling Store Buffers



`buf(x = 1)`

$$\mathsf{EN} = \left\{ \begin{array}{l} (1, \{W_\tau(), t1 = y\}), \\ (2, \{y = 1\}) \end{array} \right\}$$

t_1 `= y`

$$\mathsf{EN} = \left\{ \begin{array}{l} (1, \{W_\tau()\}), \\ (2, \{y = 1\}) \end{array} \right\}$$

`buf(y = 0)`

$$\mathsf{EN} = \left\{ \begin{array}{l} (1, \{W_\tau()\}), \\ (2, \{W_\tau(), t_2 = x\}) \end{array} \right\}$$

t_2 `= x`

$$\mathsf{EN} = \{\dots\}$$
$$\mathsf{BT} = \{(1, \{W_\tau()\})\}$$

$W_\tau()$

$$\mathsf{EN} = \{\dots\}$$

- Here is the same store buffer example we've been looking at
- We will examine one trace of its execution to show how we model store buffers
- For simplicity, we will just look at the enabled sets; the done and backtrack sets are operationally similar to before
- The writes to memory by the two threads are replaced, instead, by a thread-local buffered write
- Examining the first statement, we can see that thread one can either flush its buffer or execute its next statement
- Thread two can only execute its next statement since its buffer is empty
- At the next statement, thread one can only flush its buffer since it executed its final statement
- When thread two executes, its enabled set is analogous to thread one's: after its first statement it can either flush its buffer or execute its next statement. For the sake of space, I hid the actual contents of the enabled sets.
- Finally, when thread one executes its final statement, the flush of its buffer, we can see there is a dependent access between thread twos read and the buffer flush
- As a result, we update the backtrack set such that the buffer flush occurs before the read of x
- While we are not going into the algorithmic details, notice how this process of identifying and flipping dependent statements is essentially the same as in the original DPOR algorithm.
- In this way, we gain all the pruning power of DPOR while considering a relaxed memory model

Intra-thread Program Order Relaxations

Thread One
`x = 1`
t_1 = `y`

- Next, I'll provide a quick example of how we relax the intra-thread program order
- As I will show, the key issue with this is the lack of observability within a dynamic analysis
- Consider this fragment of the program we looked at earlier
- Under TSO, the write and the subsequent read can be re-ordered
- However, since our analysis is dynamic, we are unable to see that these two statements are adjacent and can be re-ordered until after they are executed.
- As such, we have to dynamically update the enabled set. Consider the following example:
- During the analysis, we can see the point right before thread one executes its write to `x`
- After executing this statement, we can then see the point right before it executes its read to `y`
- It is only at this point we can see that the enabled set of the previous statement should include the read to `y`
- As such, we go backwards and dynamically update the enabled set of the first statement.
- This gives us the correct possibility of re-ordering these two statements within the thread
- Just to reiterate, we can see again here how the framework is able to reduce the number of tested thread schedules.
- Although the result of this process enables an additional statement, the statement will only be executed if it conflicts with another statement across threads
- Additionally, by performing this analysis dynamically we have a significant gain in acurracy over, say, a static analysis
- A static analysis will necessairly over-approximate the enabled set resulting in decreases in acurracy and increases in runtime
- In essense, dynamically updating the enabled set provides similar accurracy increases as moving from the original static partial-order reduction algorithm to the powerful dynamic partial order reduction algorithm

Intra-thread Program Order Relaxations

Thread One
`x = 1`
t_1 `= y`

init
EN $= \{$`x = 1`$\}$

- Next, I'll provide a quick example of how we relax the intra-thread program order
- As I will show, the key issue with this is the lack of observability within a dynamic analysis
- Consider this fragment of the program we looked at earlier
- Under TSO, the write and the subsequent read can be re-ordered
- However, since our analysis is dynamic, we are unable to see that these two statements are adjacent and can be re-ordered until after they are executed.
- As such, we have to dynamically update the enabled set. Consider the following example:
- During the analysis, we can see the point right before thread one executes its write to `x`
- After executing this statement, we can then see the point right before it executes its read to `y`
- It is only at this point we can see that the enabled set of the previous statement should include the read to `y`
- As such, we go backwards and dynamically update the enabled set of the first statement.
- This gives us the correct possibility of re-ordering these two statements within the thread
- Just to reiterate, we can see again here how the framework is able to reduce the number of tested thread schedules.
- Although the result of this process enables an additional statement, the statement will only be executed if it conflicts with another statement across threads
- Additionally, by performing this analysis dynamically we have a significant gain in acurracy over, say, a static analysis
- A static analysis will necessairly over-approximate the enabled set resulting in decreases in acurracy and increases in runtime
- In essense, dynamically updating the enabled set provides similar accurracy increases as moving from the original static partial-order reduction algorithm to the powerful dynamic partial order reduction algorithm

Intra-thread Program Order Relaxations

Thread One
$\texttt{x = 1}$
t_1 $\texttt{= y}$

init

$\text{EN} = \{\texttt{x = 1}\}$

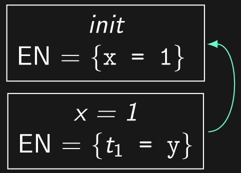
$x = 1$

$\text{EN} = \{t_1 \texttt{ = y}\}$

- Next, I'll provide a quick example of how we relax the intra-thread program order
- As I will show, the key issue with this is the lack of observability within a dynamic analysis
- Consider this fragment of the program we looked at earlier
- Under TSO, the write and the subsequent read can be re-ordered
- However, since our analysis is dynamic, we are unable to see that these two statements are adjacent and can be re-ordered until after they are executed.
- As such, we have to dynamically update the enabled set. Consider the following example:
- During the analysis, we can see the point right before thread one executes its write to x
- After executing this statement, we can then see the point right before it executes its read to y
- It is only at this point we can see that the enabled set of the previous statement should include the read to y
- As such, we go backwards and dynamically update the enabled set of the first statement.
- This gives us the correct possibility of re-ordering these two statements within the thread
- Just to reiterate, we can see again here how the framework is able to reduce the number of tested thread schedules.
- Although the result of this process enables an additional statement, the statement will only be executed if it conflicts with another statement across threads
- Additionally, by performing this analysis dynamically we have a significant gain in acurracy over, say, a static analysis
- A static analysis will necessairly over-approximate the enabled set resulting in decreases in acurracy and increases in runtime
- In essense, dynamically updating the enabled set provides similar accurracy increases as moving from the original static partial-order reduction algorithm to the powerful dynamic partial order reduction algorithm

Intra-thread Program Order Relaxations

Thread One
`x = 1`
t_1 `= y`



init

EN $= \{$`x = 1`$\}$

x = 1

EN $= \{t_1$ `= y`$\}$

- Next, I'll provide a quick example of how we relax the intra-thread program order
- As I will show, the key issue with this is the lack of observability within a dynamic analysis
- Consider this fragment of the program we looked at earlier
- Under TSO, the write and the subsequent read can be re-ordered
- However, since our analysis is dynamic, we are unable to see that these two statements are adjacent and can be re-ordered until after they are executed.
- As such, we have to dynamically update the enabled set. Consider the following example:
- During the analysis, we can see the point right before thread one executes its write to `x`
- After executing this statement, we can then see the point right before it executes its read to `y`
- It is only at this point we can see that the enabled set of the previous statement should include the read to `y`
- As such, we go backwards and dynamically update the enabled set of the first statement.
- This gives us the correct possibility of re-ordering these two statements within the thread
- Just to reiterate, we can see again here how the framework is able to reduce the number of tested thread schedules.
- Although the result of this process enables an additional statement, the statement will only be executed if it conflicts with another statement across threads
- Additionally, by performing this analysis dynamically we have a significant gain in acurracy over, say, a static analysis
- A static analysis will necessairly over-approximate the enabled set resulting in decreases in acurracy and increases in runtime
- In essense, dynamically updating the enabled set provides similar accurracy increases as moving from the original static partial-order reduction algorithm to the powerful dynamic partial order reduction algorithm

Intra-thread Program Order Relaxations

Thread One
`x = 1`
t_1 `= y`

init
EN = {`x = 1`, t_1 `= y`}

x = 1
EN = {t_1 `= y`}

- Next, I'll provide a quick example of how we relax the intra-thread program order
- As I will show, the key issue with this is the lack of observability within a dynamic analysis
- Consider this fragment of the program we looked at earlier
- Under TSO, the write and the subsequent read can be re-ordered
- However, since our analysis is dynamic, we are unable to see that these two statements are adjacent and can be re-ordered until after they are executed.
- As such, we have to dynamically update the enabled set. Consider the following example:
- During the analysis, we can see the point right before thread one executes its write to `x`
- After executing this statement, we can then see the point right before it executes its read to `y`
- It is only at this point we can see that the enabled set of the previous statement should include the read to `y`
- As such, we go backwards and dynamically update the enabled set of the first statement.
- This gives us the correct possibility of re-ordering these two statements within the thread
- Just to reiterate, we can see again here how the framework is able to reduce the number of tested thread schedules.
- Although the result of this process enables an additional statement, the statement will only be executed if it conflicts with another statement across threads
- Additionally, by performing this analysis dynamically we have a significant gain in acurracy over, say, a static analysis
- A static analysis will necessairly over-approximate the enabled set resulting in decreases in acurracy and increases in runtime
- In essense, dynamically updating the enabled set provides similar accurracy increases as moving from the original static partial-order reduction algorithm to the powerful dynamic partial order reduction algorithm

Introduction

Background

Relaxed Memory Models

Stateless Model Checking

rInspect

Experiments

Conclusion

Next, I'll go over our experimental results

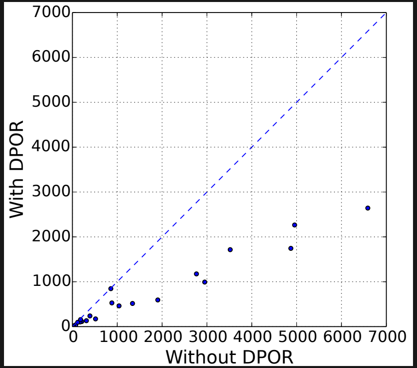
rInspect

- LLVM dynamic analysis
- C/C++ PThread Programs
- Litmus programs, SVCOMP

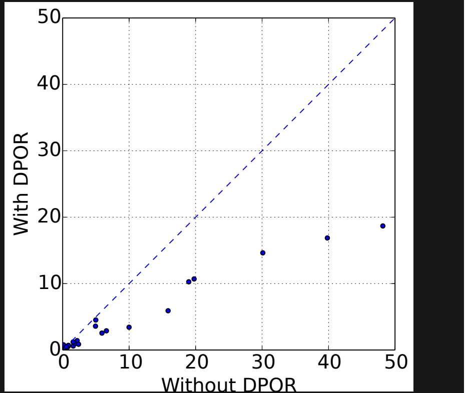
- We implemented our tool, rInspect, as a dynamic analysis framework using LLVM for instrumentation
- Our target was on C/C++ programs written using pthreads
- We evaluated our approach on litmus test programs provided by hardware vendors as well as programs from the iternational software verification competition

DPOR Results

Number of Runs



Runtime



- The figures on this slide summarize the results of applying DPOR to testing a relaxed memory model
- On the X axis, we have the results without using DPOR and on the Y axis we have the results using DPOR
- As we can see, the results all fall in the bottom right section showing that DPOR can offer a reduction
- Similarl results, in the past, have been shown for sequentially consistent programs
- As expected, these results also carry over to programs tested under a weak memory model
- These two figures are for TSO, but the results for PSO are similar. They can be found in our paper.

Litmus Tests

Method	Passed	Failed	Avg. # Runs

- Here are the results of running our tool on the litmus bench programs
- These programs were provided by the hardware vendors as examples of acceptable behavior under their relaxed architectures.
- If the test exposes relaxed behavior then it will fail, otherwise it will pass
- First, under SC, we can see the verifier does not find any relaxed behavior
- We normalized the number of runs with respect to the sequentially consistent tests
- Under TSO, we can see the number of failing runs increases. This means the verifier explores more of the relaxed memory behavior
- Additionally, the overhead moving from SC to TSO is about 5 times
- PSO, which is more relaxed than TSO, explores more relaxed behavior and has an even higher overhead
- Also, notice that all of these tests are using DPOR. As such, the total number of runs in the fully enumerated space is much higher
- Additionally, even though the overhead in terms of number of runs increases, the SC results with respect to a TSO or PSO architecture are incorrect: the verifier misses the relaxed behavior

Litmus Tests

Method	Passed	Failed	Avg. # Runs
DPOR$_{SC}$	121	0	1.0 X

- Here are the results of running our tool on the litmus bench programs
- These programs were provided by the hardware vendors as examples of acceptable behavior under their relaxed architectures.
- If the test exposes relaxed behavior then it will fail, otherwise it will pass
- First, under SC, we can see the verifier does not find any relaxed behavior
- We normalized the number of runs with respect to the sequentially consistent tests
- Under TSO, we can see the number of failing runs increases. This means the verifier explores more of the relaxed memory behavior
- Additionally, the overhead moving from SC to TSO is about 5 times
- PSO, which is more relaxed than TSO, explores more relaxed behavior and has an even higher overhead
- Also, notice that all of these tests are using DPOR. As such, the total number of runs in the fully enumerated space is much higher
- Additionally, even though the overhead in terms of number of runs increases, the SC results with respect to a TSO or PSO architecture are incorrect: the verifier misses the relaxed behavior

Litmus Tests

Method	Passed	Failed	Avg. # Runs
DPOR_{SC}	121	0	1.0 X
DPOR_{TSO}	47	73	5.0 X

- Here are the results of running our tool on the litmus bench programs
- These programs were provided by the hardware vendors as examples of acceptable behavior under their relaxed architectures.
- If the test exposes relaxed behavior then it will fail, otherwise it will pass
- First, under SC, we can see the verifier does not find any relaxed behavior
- We normalized the number of runs with respect to the sequentially consistent tests
- Under TSO, we can see the number of failing runs increases. This means the verifier explores more of the relaxed memory behavior
- Additionally, the overhead moving from SC to TSO is about 5 times
- PSO, which is more relaxed than TSO, explores more relaxed behavior and has an even higher overhead
- Also, notice that all of these tests are using DPOR. As such, the total number of runs in the fully enumerated space is much higher
- Additionally, even though the overhead in terms of number of runs increases, the SC results with respect to a TSO or PSO architecture are incorrect: the verifier misses the relaxed behavior

Litmus Tests

Method	Passed	Failed	Avg. # Runs
DPOR_{SC}	121	0	1.0 X
DPOR_{TSO}	47	73	5.0 X
DPOR_{PSO}	24	97	11.0 X

- Here are the results of running our tool on the litmus bench programs
- These programs were provided by the hardware vendors as examples of acceptable behavior under their relaxed architectures.
- If the test exposes relaxed behavior then it will fail, otherwise it will pass
- First, under SC, we can see the verifier does not find any relaxed behavior
- We normalized the number of runs with respect to the sequentially consistent tests
- Under TSO, we can see the number of failing runs increases. This means the verifier explores more of the relaxed memory behavior
- Additionally, the overhead moving from SC to TSO is about 5 times
- PSO, which is more relaxed than TSO, explores more relaxed behavior and has an even higher overhead
- Also, notice that all of these tests are using DPOR. As such, the total number of runs in the fully enumerated space is much higher
- Additionally, even though the overhead in terms of number of runs increases, the SC results with respect to a TSO or PSO architecture are incorrect: the verifier misses the relaxed behavior

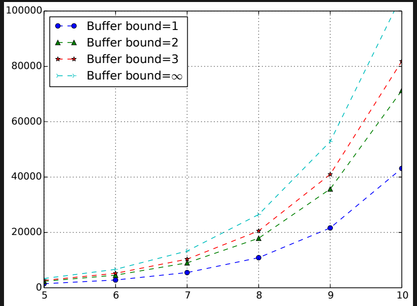
Conclusion

- rInspect: DPOR for RMM
- Dynamic verification
- Unified treatment of SC and RMM

- In conclusion, we prested rInspect, a DPOR implementation for relaxed memory models
- It performs dynamic verification of multithreaded programs using stateless model checking
- As we showed, it unifies the treatment of sequentially consistent and relaxed behaviors in multithreaded programs
- As a result, it is capable of offering a significant runtime reduction compared to a full enumeration of the state space
- Also, I skipped over a heuristic exploration technique called buffer bounding, which is askin to preemptive context bounding for weak memory.
- You can find the details in our paper
- With that, I'll take any questions

Conclusion

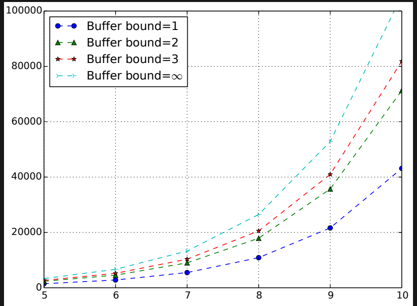
- rInspect: DPOR for RMM
- Dynamic verification
- Unified treatment of SC and RMM



- In conclusion, we prested rInspect, a DPOR implementation for relaxed memory models
- It performs dynamic verification of multithreaded programs using stateless model checking
- As we showed, it unifies the treatment of sequentially consistent and relaxed behaviors in multithreaded programs
- As a result, it is capable of offering a significant runtime reduction compared to a full enumeration of the state space
- Also, I skipped over a heuristic exploration technique called buffer bounding, which is askin to preemptive context bounding for weak memory.
- You can find the details in our paper
- With that, I'll take any questions

Conclusion

- rInspect: DPOR for RMM
- Dynamic verification
- Unified treatment of SC and RMM



Questions?

- In conclusion, we prested rInspect, a DPOR implementation for relaxed memory models
- It performs dynamic verification of multithreaded programs using stateless model checking
- As we showed, it unifies the treatment of sequentially consistent and relaxed behaviors in multithreaded programs
- As a result, it is capable of offering a significant runtime reduction compared to a full enumeration of the state space
- Also, I skipped over a heuristic exploration technique called buffer bounding, which is askin to preemptive context bounding for weak memory.
- You can find the details in our paper
- With that, I'll take any questions