Robustness against Power is PSpace-complete

Egor Derevenetc[1,2] Roland Meyer[1]

[1]University of Kaiserslautern

[2]Fraunhofer ITWM

WEACON
Kaiserslautern
13.06.2014

Introduction

Power Architecture

Robustness

Deciding Robustness

Characterization of Violating Computations

Normal-Form Computations

Generating Normal-Form Computations

Checking Cyclicity of Happens-Before Relation

Complexity

Conclusion

Related Work

Summary

Power Architecture 1/4

Example (Message Passing Program)

Consider the multithreaded program (initially, $x = y = 0$):

$$
\begin{array}{c||c}
\text{Thread 1:} & \text{Thread 2:} \\
a\colon \texttt{mem}[x] \leftarrow 1 & c\colon \mathrm{r}_1 \leftarrow \texttt{mem}[y] \\
b\colon \texttt{mem}[y] \leftarrow 1 & d\colon \mathrm{r}_2 \leftarrow \texttt{mem}[x]
\end{array}
$$

Assumption: $\mathrm{r}_1 = 1$ implies $\mathrm{r}_2 = 1$.

Power Architecture 1/4

Example (Message Passing Program)

Consider the multithreaded program (initially, $x = y = 0$):

$$\begin{array}{c|c} \text{Thread 1:} & \text{Thread 2:} \\ a: \texttt{mem[}x\texttt{]} \leftarrow 1 & c: \mathrm{r}_1 \leftarrow \texttt{mem[}y\texttt{]} \\ b: \texttt{mem[}y\texttt{]} \leftarrow 1 & d: \mathrm{r}_2 \leftarrow \texttt{mem[}x\texttt{]} \end{array}$$

Assumption: $\mathrm{r}_1 = 1$ implies $\mathrm{r}_2 = 1$.

Sequential Consistency (SC) [Lamport, 1979]

- Instructions are executed in order.
- Writes to memory are immediately visible to all threads.

$\Rightarrow$ The assumption holds.

Power Architecture 1/4

Example (Message Passing Program)

Consider the multithreaded program (initially, $x = y = 0$):

$$\begin{array}{c||c} \text{Thread 1:} & \text{Thread 2:} \\ a\colon \mathtt{mem}[x] \leftarrow 1 & c\colon \mathrm{r}_1 \leftarrow \mathtt{mem}[y] \\ b\colon \mathtt{mem}[y] \leftarrow 1 & d\colon \mathrm{r}_2 \leftarrow \mathtt{mem}[x] \end{array}$$

Assumption: $\mathrm{r}_1 = 1$ implies $\mathrm{r}_2 = 1$.

Sequential Consistency (SC) [Lamport, 1979]

- Instructions are executed in order.
- Writes to memory are immediately visible to all threads.

$\Rightarrow$ The assumption holds.

Power Architecture by IBM et al. [Sarkar et al., 2011]

- Independent instructions can be executed out of order.
- Writes can be seen by different threads in different order.

$\Rightarrow$ The assumption does not hold.

Power Architecture 2/4

How a thread executes an instruction on Power:

Power Architecture 2/4

How a thread executes an instruction on Power:

- First, it fetches it. Instructions must be fetched in the program order, one after another.

Power Architecture 2/4

How a thread executes an instruction on Power:

- First, it fetches it. Instructions must be fetched in the program order, one after another.
- Next, it performs the computation prescribed by the instruction's semantics. Results of instructions, on which the current one depends, must be already computed.

Power Architecture 2/4

How a thread executes an instruction on Power:

- First, it fetches it. Instructions must be fetched in the program order, one after another.
- Next, it performs the computation prescribed by the instruction's semantics. Results of instructions, on which the current one depends, must be already computed.
- Finally, it commits the instruction. Similarly, all instruction's dependencies must be committed earlier.

Power Architecture 2/4

How a thread executes an instruction on Power:

- First, it fetches it. Instructions must be fetched in the program order, one after another.
- Next, it performs the computation prescribed by the instruction's semantics. Results of instructions, on which the current one depends, must be already computed.
- Finally, it commits the instruction. Similarly, all instruction's dependencies must be committed earlier.

One thread can execute multiple instructions in parallel.

Power Architecture 2/4

How a thread executes an instruction on Power:

- First, it fetches it. Instructions must be fetched in the program order, one after another.
- Next, it performs the computation prescribed by the instruction's semantics. Results of instructions, on which the current one depends, must be already computed.
- Finally, it commits the instruction. Similarly, all instruction's dependencies must be committed earlier.

One thread can execute multiple instructions in parallel.

Example (Thread 2 of Message Passing Program)

$c : \mathsf{r}_1 \leftarrow \mathtt{mem}[y];\ d : \mathsf{r}_2 \leftarrow \mathtt{mem}[x].$

Power Architecture 2/4

How a thread executes an instruction on Power:

- First, it fetches it. Instructions must be fetched in the program order, one after another.
- Next, it performs the computation prescribed by the instruction's semantics. Results of instructions, on which the current one depends, must be already computed.
- Finally, it commits the instruction. Similarly, all instruction's dependencies must be committed earlier.

One thread can execute multiple instructions in parallel.

Example (Thread 2 of Message Passing Program)

$c: \mathsf{r}_1 \leftarrow \mathtt{mem}[y];\ d: \mathsf{r}_2 \leftarrow \mathtt{mem}[x].$

Example (Computation of Thread 2)

$\beta := \mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{load}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d) \cdot \mathsf{commit}(c).$

Power Architecture 3/4

How memory works on Power:

Power Architecture 3/4

How memory works on Power:

- A thread loads the value written by the last store to the same address propagated to this thread.

Power Architecture 3/4

How memory works on Power:

- A thread loads the value written by the last store to the same address propagated to this thread.
- A committed store is immediately propagated to its own thread and can be later propagated to some other threads.

Power Architecture 3/4

How memory works on Power:

- A thread loads the value written by the last store to the same address propagated to this thread.
- A committed store is immediately propagated to its own thread and can be later propagated to some other threads.
- Stores to the same address are globally ordered (coherence order) and can be propagated only in this order.

Power Architecture 3/4

How memory works on Power:

- A thread loads the value written by the last store to the same address propagated to this thread.
- A committed store is immediately propagated to its own thread and can be later propagated to some other threads.
- Stores to the same address are globally ordered (coherence order) and can be propagated only in this order.

Example (Thread 1 of Message Passing Program)

a: $\texttt{mem}[x] \leftarrow 1$; b: $\texttt{mem}[y] \leftarrow 1$.

Power Architecture 3/4

How memory works on Power:

- A thread loads the value written by the last store to the same address propagated to this thread.
- A committed store is immediately propagated to its own thread and can be later propagated to some other threads.
- Stores to the same address are globally ordered (coherence order) and can be propagated only in this order.

Example (Thread 1 of Message Passing Program)

a: $\texttt{mem}[x] \leftarrow 1$; b: $\texttt{mem}[y] \leftarrow 1$.

Example (Computation of Thread 1)

$\alpha := \mathsf{fetch}(a) \cdot \mathsf{commit}(a) \cdot \mathsf{prop}(a, 1) \cdot \mathsf{fetch}(b) \cdot \mathsf{commit}(b) \cdot \mathsf{prop}(b, 1) \cdot \mathsf{prop}(b, 2)$.

Power Architecture 4/4

Example (Message Passing Program)

$$\text{Initially, } x = y = 0.$$

$$\begin{array}{c||c} \text{Thread 1:} & \text{Thread 2:} \\ a: \texttt{mem}[x] \leftarrow 1 & c: \mathsf{r}_1 \leftarrow \texttt{mem}[y] \\ b: \texttt{mem}[y] \leftarrow 1 & d: \mathsf{r}_2 \leftarrow \texttt{mem}[x] \end{array}$$

Assumption: $\mathsf{r}_1 = 1$ implies $\mathsf{r}_2 = 1$.

Example (Computation of the Program on Power)

$\tau := \alpha \cdot \beta = \mathsf{fetch}(a) \cdot \mathsf{commit}(a) \cdot \mathsf{prop}(a, 1) \cdot \mathsf{fetch}(b) \cdot \mathsf{commit}(b) \cdot \mathsf{prop}(b, 1) \cdot \mathsf{prop}(b, 2) \cdot \mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{load}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d) \cdot \mathsf{commit}(c)$.

- Load c reads value 1 written by b.
- Load d reads the initial value 0, as store a was never propagated to Thread 2.

$\Rightarrow$ The assumption does not hold.

Robustness

Robustness Problem

Check, whether a given program has the same behaviors under SC and under Power.

Robustness

Robustness Problem

Check, whether a given program has the same behaviors under SC and under Power. Behavior is the control and data dependencies between instructions.

Robustness

Robustness Problem

Check, whether a given program has the same behaviors under SC and under Power. Behavior is the control and data dependencies between instructions.

Our Solution

Reduce robustness checking to an emptiness check for an intersection of languages:

$$\mathcal{L} \cap \mathcal{R} \stackrel{?}{=} \emptyset.$$

- Computations violating SC (if any) have a representative in a normal form.

Robustness

Robustness Problem

Check, whether a given program has the same behaviors under SC and under Power. Behavior is the control and data dependencies between instructions.

Our Solution

Reduce robustness checking to an emptiness check for an intersection of languages:

$$\mathcal{L} \cap \mathcal{R} \stackrel{?}{=} \emptyset.$$

- Computations violating SC (if any) have a representative in a normal form.
- Language $\mathcal{L}$ consists of all normal-form computations.

Robustness

Robustness Problem

Check, whether a given program has the same behaviors under SC and under Power. Behavior is the control and data dependencies between instructions.

Our Solution

Reduce robustness checking to an emptiness check for an intersection of languages:

$$\mathcal{L} \cap \mathcal{R} \stackrel{?}{=} \emptyset.$$

- Computations violating SC (if any) have a representative in a normal form.
- Language $\mathcal{L}$ consists of all normal-form computations.
- $\cap \mathcal{R}$ filters only violating computations.

Robustness

Robustness Problem

Check, whether a given program has the same behaviors under SC and under Power. Behavior is the control and data dependencies between instructions.

Our Solution

Reduce robustness checking to an emptiness check for an intersection of languages:

$$\mathcal{L} \cap \mathcal{R} \stackrel{?}{=} \emptyset.$$

- Computations violating SC (if any) have a representative in a normal form.
- Language $\mathcal{L}$ consists of all normal-form computations.
- $\cap \mathcal{R}$ filters only violating computations.
- Decide $\mathcal{L} \cap \mathcal{R} \stackrel{?}{=} \emptyset$.

Introduction
- Power Architecture
- Robustness

Deciding Robustness
- Characterization of Violating Computations
- Normal-Form Computations
- Generating Normal-Form Computations
- Checking Cyclicity of Happens-Before Relation
- Complexity

Conclusion
- Related Work
- Summary

Introduction

Power Architecture

Robustness

Deciding Robustness

Characterization of Violating Computations

Normal-Form Computations

Generating Normal-Form Computations

Checking Cyclicity of Happens-Before Relation

Complexity

Conclusion

Related Work

Summary

Characterization of Violating Computations

Lemma ([Shasha and Snir, 1988])

A computation violates SC iff it has cyclic happens-before relation.

Characterization of Violating Computations

Lemma ([Shasha and Snir, 1988])

A computation violates SC iff it has cyclic happens-before relation.

Example (Happens-Before Relation of Computation τ)

	Thread 1	Thread 2
init_x	a: `mem`[x] $\leftarrow 1$	d: $\mathrm{r}_2 \leftarrow$ `mem`[x]
init_y	b: `mem`[y] $\leftarrow 1$	c: $\mathrm{r}_1 \leftarrow$ `mem`[y]

Happens-before relation is a union of four relations:

Characterization of Violating Computations

Lemma ([Shasha and Snir, 1988])

A computation violates SC iff it has cyclic happens-before relation.

Example (Happens-Before Relation of Computation τ)

$$
\begin{array}{lcc}
 & \text{Thread 1} & \text{Thread 2} \\
\text{init}_x & a:\ \mathtt{mem}[x] \leftarrow 1 & d:\ \mathrm{r}_2 \leftarrow \mathtt{mem}[x] \\
 & po\downarrow & po\uparrow \\
\text{init}_y & b:\ \mathtt{mem}[y] \leftarrow 1 & c:\ \mathrm{r}_1 \leftarrow \mathtt{mem}[y]
\end{array}
$$

Happens-before relation is a union of four relations:

- Program order — textual ordering of instructions.

Characterization of Violating Computations

Lemma ([Shasha and Snir, 1988])

A computation violates SC iff it has cyclic happens-before relation.

Example (Happens-Before Relation of Computation τ)

	Thread 1	Thread 2
$\text{init}_x \xrightarrow{co}$	a: $\texttt{mem}[x] \leftarrow 1$	d: $\mathrm{r}_2 \leftarrow \texttt{mem}[x]$
	$po \downarrow$	$po \uparrow$
$\text{init}_y \xrightarrow{co}$	b: $\texttt{mem}[y] \leftarrow 1$	c: $\mathrm{r}_1 \leftarrow \texttt{mem}[y]$

Happens-before relation is a union of four relations:

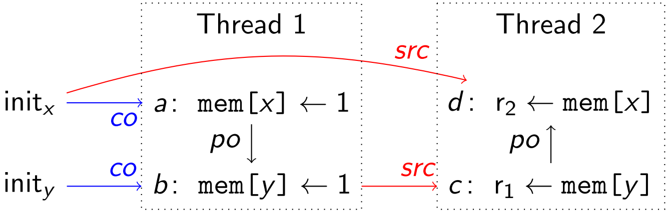
- Program order — textual ordering of instructions.
- Coherence order — ordering of stores to the same address.

Characterization of Violating Computations

Lemma ([Shasha and Snir, 1988])

A computation violates SC iff it has cyclic happens-before relation.

Example (Happens-Before Relation of Computation τ)



Happens-before relation is a union of four relations:

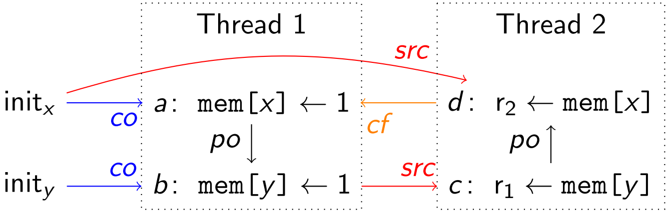
- Program order — textual ordering of instructions.
- Coherence order — ordering of stores to the same address.
- Source order — which store is read by which load.

Characterization of Violating Computations

Lemma ([Shasha and Snir, 1988])

A computation violates SC iff it has cyclic happens-before relation.

Example (Happens-Before Relation of Computation τ)



Happens-before relation is a union of four relations:

- Program order — textual ordering of instructions.
- Coherence order — ordering of stores to the same address.
- Source order — which store is read by which load.
- Conflict order — which stores overwrite the value read by a load.

Introduction

Power Architecture

Robustness

Deciding Robustness

Characterization of Violating Computations

Normal-Form Computations

Generating Normal-Form Computations

Checking Cyclicity of Happens-Before Relation

Complexity

Conclusion

Related Work

Summary

Normal-Form Computations 1/4

Definition

A computation $\tau := \tau_1 \cdots \tau_n$ is in normal form of degree n, if

Normal-Form Computations 1/4

Definition

A computation $\tau := \tau_1 \cdots \tau_n$ is in normal form of degree n, if

- there are no fetch events in $\tau_2 \cdots \tau_n$,

Normal-Form Computations 1/4

Definition

A computation $\tau := \tau_1 \cdots \tau_n$ is in normal form of degree n, if

- there are no fetch events in $\tau_2 \cdots \tau_n$,
- events in each part $\tau_1 \ldots \tau_n$ occur in the order in which corresponding fetch events occur in τ_1.

Normal-Form Computations 1/4

Definition

A computation $\tau := \tau_1 \cdots \tau_n$ is in normal form of degree n, if

- there are no fetch events in $\tau_2 \cdots \tau_n$,
- events in each part $\tau_1 \ldots \tau_n$ occur in the order in which corresponding fetch events occur in τ_1.

Theorem

If a program has computations with cyclic happens-before relation, it has one in the normal form of degree (number of threads + 3).

Normal-Form Computations 1/4

Definition

A computation $\tau := \tau_1 \cdots \tau_n$ is in normal form of degree n, if

- there are no fetch events in $\tau_2 \cdots \tau_n$,
- events in each part $\tau_1 \ldots \tau_n$ occur in the order in which corresponding fetch events occur in τ_1.

Theorem

If a program has computations with cyclic happens-before relation, it has one in the normal form of degree (number of threads + 3).

Proof Idea.

Take a shortest computation with cyclic happens-before relation and transform it to the normal form. $\square$

Normal-Form Computations 2/4

Lemma

Given a non-empty valid computation, there is a thread, such that deletion of all events belonging to its last fetched instruction produces a valid computation.

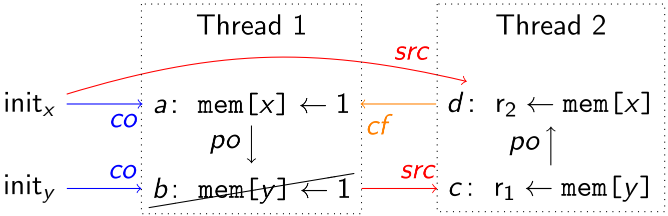
Normal-Form Computations 2/4

Lemma

Given a non-empty valid computation, there is a thread, such that deletion of all events belonging to its last fetched instruction produces a valid computation.

Example

$\tau = \mathsf{fetch}(a) \cdot \mathsf{commit}(a) \cdot \mathsf{prop}(a, 1) \cdot \cancel{\mathsf{fetch}(b)} \cdot \cancel{\mathsf{commit}(b)} \cdot \cancel{\mathsf{prop}(b, 1)} \cdot \cancel{\mathsf{prop}(b, 2)} \cdot \mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{load}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d) \cdot \mathsf{commit}(c)$.



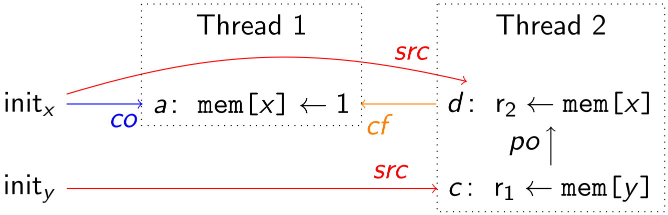
Normal-Form Computations 2/4

Lemma

Given a non-empty valid computation, there is a thread, such that deletion of all events belonging to its last fetched instruction produces a valid computation.

Example

$\tau' = \mathsf{fetch}(a) \cdot \mathsf{commit}(a) \cdot \mathsf{prop}(a, 1)$
$\cdot\ \mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{load}(c) \cdot \mathsf{load}(d) \cdot$
$\mathsf{commit}(d) \cdot \mathsf{commit}(c)$.



Normal-Form Computations 3/4

1. Let $\tau := \tau_1 \cdot e_1 \cdot \tau_2 \cdot e_2 \cdots \tau_n$ be a shortest computation with cyclic happens-before relation.

Normal-Form Computations 3/4

1. Let $\tau := \tau_1 \cdot e_1 \cdot \tau_2 \cdot e_2 \cdots \tau_n$ be a shortest computation with cyclic happens-before relation.
2. Let $e_1 \cdots e_{n-1}$ be the deleted events.

Normal-Form Computations 3/4

1. Let $\tau := \tau_1 \cdot e_1 \cdot \tau_2 \cdot e_2 \cdots \tau_n$ be a shortest computation with cyclic happens-before relation.
2. Let $e_1 \cdots e_{n-1}$ be the deleted events.
3. Assume all fetch events are in $\tau_1 \cdot e_1$ (one can always move them to the front).

Normal-Form Computations 3/4

1. Let $\tau := \tau_1 \cdot e_1 \cdot \tau_2 \cdot e_2 \cdots \tau_n$ be a shortest computation with cyclic happens-before relation.
2. Let $e_1 \cdots e_{n-1}$ be the deleted events.
3. Assume all fetch events are in $\tau_1 \cdot e_1$ (one can always move them to the front).
4. Computation $\tau' := \tau_1 \cdot \tau_2 \cdots \tau_n$ is shorter than τ

Normal-Form Computations 3/4

1. Let $\tau := \tau_1 \cdot e_1 \cdot \tau_2 \cdot e_2 \cdots \tau_n$ be a shortest computation with cyclic happens-before relation.
2. Let $e_1 \cdots e_{n-1}$ be the deleted events.
3. Assume all fetch events are in $\tau_1 \cdot e_1$ (one can always move them to the front).
4. Computation $\tau' := \tau_1 \cdot \tau_2 \cdots \tau_n$ is shorter than τ,
   $\Rightarrow$ not violating.

Normal-Form Computations 3/4

1. Let $\tau := \tau_1 \cdot e_1 \cdot \tau_2 \cdot e_2 \cdots \tau_n$ be a shortest computation with cyclic happens-before relation.
2. Let $e_1 \cdots e_{n-1}$ be the deleted events.
3. Assume all fetch events are in $\tau_1 \cdot e_1$ (one can always move them to the front).
4. Computation $\tau' := \tau_1 \cdot \tau_2 \cdots \tau_n$ is shorter than τ,
   $\Rightarrow$ not violating.
5. Let σ be a sequentially consistent version of τ'.

Normal-Form Computations 3/4

1. Let $\tau := \tau_1 \cdot e_1 \cdot \tau_2 \cdot e_2 \cdots \tau_n$ be a shortest computation with cyclic happens-before relation.
2. Let $e_1 \cdots e_{n-1}$ be the deleted events.
3. Assume all fetch events are in $\tau_1 \cdot e_1$ (one can always move them to the front).
4. Computation $\tau' := \tau_1 \cdot \tau_2 \cdots \tau_n$ is shorter than τ, $\Rightarrow$ not violating.
5. Let σ be a sequentially consistent version of τ'.
6. Reorder events in the way they follow in σ:

$$\tau'' := \sigma{\downarrow}\tau_1 \cdot e_1 \cdots \sigma{\downarrow}\tau_n.$$

Normal-Form Computations 3/4

1. Let $\tau := \tau_1 \cdot e_1 \cdot \tau_2 \cdot e_2 \cdots \tau_n$ be a shortest computation with cyclic happens-before relation.
2. Let $e_1 \cdots e_{n-1}$ be the deleted events.
3. Assume all fetch events are in $\tau_1 \cdot e_1$ (one can always move them to the front).
4. Computation $\tau' := \tau_1 \cdot \tau_2 \cdots \tau_n$ is shorter than τ,
   $\Rightarrow$ not violating.
5. Let σ be a sequentially consistent version of τ'.
6. Reorder events in the way they follow in σ:

$$\tau'' := \sigma \downarrow \tau_1 \cdot e_1 \cdots \sigma \downarrow \tau_n.$$

Lemma

Computation τ''

- *is in normal form,*

Normal-Form Computations 3/4

1. Let $\tau := \tau_1 \cdot e_1 \cdot \tau_2 \cdot e_2 \cdots \tau_n$ be a shortest computation with cyclic happens-before relation.
2. Let $e_1 \cdots e_{n-1}$ be the deleted events.
3. Assume all fetch events are in $\tau_1 \cdot e_1$ (one can always move them to the front).
4. Computation $\tau' := \tau_1 \cdot \tau_2 \cdots \tau_n$ is shorter than τ,
   $\Rightarrow$ not violating.
5. Let σ be a sequentially consistent version of τ'.
6. Reorder events in the way they follow in σ:

$$\tau'' := \sigma{\downarrow}\tau_1 \cdot e_1 \cdots \sigma{\downarrow}\tau_n.$$

Lemma

Computation τ''

- *is in normal form,*
- *has the same happens-before relation as τ*

Normal-Form Computations 3/4

1. Let $\tau := \tau_1 \cdot e_1 \cdot \tau_2 \cdot e_2 \cdots \tau_n$ be a shortest computation with cyclic happens-before relation.
2. Let $e_1 \cdots e_{n-1}$ be the deleted events.
3. Assume all fetch events are in $\tau_1 \cdot e_1$ (one can always move them to the front).
4. Computation $\tau' := \tau_1 \cdot \tau_2 \cdots \tau_n$ is shorter than τ, $\Rightarrow$ not violating.
5. Let σ be a sequentially consistent version of τ'.
6. Reorder events in the way they follow in σ:

$$\tau'' := \sigma\!\downarrow\!\tau_1 \cdot e_1 \cdots \sigma\!\downarrow\!\tau_n.$$

Lemma

Computation τ''

- *is in normal form,*
- *has the same happens-before relation as τ, $\Rightarrow$ violating.*

Normal-Form Computations 4/4

Example

A shortest computation with cyclic happens-before relation:

$$
\begin{aligned}
\tau \quad = \quad & (\mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a)) \cdot \cancel{\mathsf{fetch}(b)} \\
\cdot \quad & (\mathsf{commit}(a) \cdot \mathsf{prop}(a, 1)) \cdot \cancel{\mathsf{commit}(b)} \cdot \cancel{\mathsf{prop}(b, 1)} \cdot \cancel{\mathsf{prop}(b, 2)} \\
\cdot \quad & (\mathsf{load}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d) \cdot \mathsf{commit}(c))
\end{aligned}
$$

Normal-Form Computations 4/4

Example

The shortened computation:

$$
\begin{aligned}
\tau' \quad = \quad & (\mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a)) \\
\cdot \quad & (\mathsf{commit}(a) \cdot \mathsf{prop}(a, 1)) \\
\cdot \quad & (\mathsf{load}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d) \cdot \mathsf{commit}(c))
\end{aligned}
$$

Normal-Form Computations 4/4

Example

The shortened computation:

$$\begin{array}{rcl}\tau' & = & (\mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a)) \\ & \cdot & (\mathsf{commit}(a) \cdot \mathsf{prop}(a, 1)) \\ & \cdot & (\mathsf{load}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d) \cdot \mathsf{commit}(c))\end{array}$$

Matching sequentially consistent computation:

$$\begin{array}{rcl}\sigma & = & \mathsf{fetch}(c) \cdot \mathsf{load}(c) \cdot \mathsf{commit}(c) \\ & \cdot & \mathsf{fetch}(d) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d) \\ & \cdot & \mathsf{fetch}(a) \cdot \mathsf{commit}(a) \cdot \mathsf{prop}(a, 1) \cdot \mathsf{prop}(a, 2)\end{array}$$

Normal-Form Computations 4/4

Example

A shortest computation with cyclic happens-before relation:

$$\begin{array}{rcl}
\tau & = & (\mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a)) \cdot \cancel{\mathsf{fetch}(b)} \\
 & \cdot & (\mathsf{commit}(a) \cdot \mathsf{prop}(a,1)) \cdot \cancel{\mathsf{commit}(b)} \cdot \cancel{\mathsf{prop}(b,1)} \cdot \cancel{\mathsf{prop}(b,2)} \\
 & \cdot & (\mathsf{load}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d) \cdot \mathsf{commit}(c))
\end{array}$$

Matching sequentially consistent computation:

$$\begin{array}{rcl}
\sigma & = & \mathsf{fetch}(c) \cdot \mathsf{load}(c) \cdot \mathsf{commit}(c) \\
 & \cdot & \mathsf{fetch}(d) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d) \\
 & \cdot & \mathsf{fetch}(a) \cdot \mathsf{commit}(a) \cdot \mathsf{prop}(a,1) \cdot \mathsf{prop}(a,2)
\end{array}$$

Normal-form computation:

$$\begin{array}{rcl}
\tau'' & = & (\mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a)) \cdot \mathsf{fetch}(b) \\
 & \cdot & (\mathsf{commit}(a) \cdot \mathsf{prop}(a,1)) \cdot \mathsf{commit}(b) \cdot \mathsf{prop}(b,1) \cdot \mathsf{prop}(b,2) \\
 & \cdot & (\mathsf{load}(c) \cdot \mathsf{commit}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d))
\end{array}$$

Normal-Form Computations 4/4

Example

A shortest computation with cyclic happens-before relation:

$$
\begin{array}{rcl}
\tau & = & (\mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a)) \cdot \cancel{\mathsf{fetch}(b)} \\
& \cdot & (\mathsf{commit}(a) \cdot \mathsf{prop}(a,1)) \cdot \cancel{\mathsf{commit}(b)} \cdot \cancel{\mathsf{prop}(b,1)} \cdot \cancel{\mathsf{prop}(b,2)} \\
& \cdot & (\mathsf{load}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d) \cdot \mathsf{commit}(c))
\end{array}
$$

Matching sequentially consistent computation:

$$
\begin{array}{rcl}
\sigma & = & \mathsf{fetch}(c) \cdot \mathsf{load}(c) \cdot \mathsf{commit}(c) \\
& \cdot & \mathsf{fetch}(d) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d) \\
& \cdot & \mathsf{fetch}(a) \cdot \mathsf{commit}(a) \cdot \mathsf{prop}(a,1) \cdot \mathsf{prop}(a,2)
\end{array}
$$

Normal-form computation:

$$
\begin{array}{rcl}
\tau'' & = & (\mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a)) \cdot \mathsf{fetch}(b) \\
& \cdot & (\mathsf{commit}(a) \cdot \mathsf{prop}(a,1)) \cdot \mathsf{commit}(b) \cdot \mathsf{prop}(b,1) \cdot \mathsf{prop}(b,2) \\
& \cdot & (\mathsf{load}(c) \cdot \mathsf{commit}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d))
\end{array}
$$

Normal-Form Computations 4/4

Example

A shortest computation with cyclic happens-before relation:

$$\begin{aligned}
\tau \quad = \quad & (\mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a)) \cdot \cancel{\mathsf{fetch}(b)} \\
\cdot \quad & (\mathsf{commit}(a) \cdot \mathsf{prop}(a,1)) \cdot \cancel{\mathsf{commit}(b)} \cdot \cancel{\mathsf{prop}(b,1)} \cdot \cancel{\mathsf{prop}(b,2)} \\
\cdot \quad & (\mathsf{load}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d) \cdot \mathsf{commit}(c))
\end{aligned}$$

Matching sequentially consistent computation:

$$\begin{aligned}
\sigma \quad = \quad & \mathsf{fetch}(c) \cdot \mathsf{load}(c) \cdot \mathsf{commit}(c) \\
\cdot \quad & \mathsf{fetch}(d) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d) \\
\cdot \quad & \mathsf{fetch}(a) \cdot \mathsf{commit}(a) \cdot \mathsf{prop}(a,1) \cdot \mathsf{prop}(a,2)
\end{aligned}$$

Normal-form computation:

$$\begin{aligned}
\tau'' \quad = \quad & (\mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a)) \cdot \mathsf{fetch}(b) \\
\cdot \quad & (\mathsf{commit}(a) \cdot \mathsf{prop}(a,1)) \cdot \mathsf{commit}(b) \cdot \mathsf{prop}(b,1) \cdot \mathsf{prop}(b,2) \\
\cdot \quad & (\mathsf{load}(c) \cdot \mathsf{commit}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d))
\end{aligned}$$

Normal-Form Computations 4/4

Example

A shortest computation with cyclic happens-before relation:

$$\begin{aligned}\tau \quad = \quad & (\mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a)) \cdot \cancel{\mathsf{fetch}(b)} \\ \cdot \quad & (\mathsf{commit}(a) \cdot \mathsf{prop}(a,1)) \cdot \cancel{\mathsf{commit}(b)} \cdot \cancel{\mathsf{prop}(b,1)} \cdot \cancel{\mathsf{prop}(b,2)} \\ \cdot \quad & (\mathsf{load}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d) \cdot \mathsf{commit}(c))\end{aligned}$$

Matching sequentially consistent computation:

$$\begin{aligned}\sigma \quad = \quad & \mathsf{fetch}(c) \cdot \mathsf{load}(c) \cdot \mathsf{commit}(c) \\ \cdot \quad & \mathsf{fetch}(d) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d) \\ \cdot \quad & \mathsf{fetch}(a) \cdot \mathsf{commit}(a) \cdot \mathsf{prop}(a,1) \cdot \mathsf{prop}(a,2)\end{aligned}$$

Normal-form computation:

$$\begin{aligned}\tau'' \quad = \quad & (\mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a)) \cdot \mathsf{fetch}(b) \\ \cdot \quad & (\mathsf{commit}(a) \cdot \mathsf{prop}(a,1)) \cdot \mathsf{commit}(b) \cdot \mathsf{prop}(b,1) \cdot \mathsf{prop}(b,2) \\ \cdot \quad & (\mathsf{load}(c) \cdot \mathsf{commit}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d))\end{aligned}$$

Normal-Form Computations 4/4

Example

A shortest computation with cyclic happens-before relation:

$$
\begin{aligned}
\tau \quad = \quad & (\mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a)) \cdot \cancel{\mathsf{fetch}(b)} \\
\cdot \quad & (\mathsf{commit}(a) \cdot \mathsf{prop}(a, 1)) \cdot \cancel{\mathsf{commit}(b)} \cdot \cancel{\mathsf{prop}(b, 1)} \cdot \cancel{\mathsf{prop}(b, 2)} \\
\cdot \quad & (\mathsf{load}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d) \cdot \mathsf{commit}(c))
\end{aligned}
$$

Matching sequentially consistent computation:

$$
\begin{aligned}
\sigma \quad = \quad & \mathsf{fetch}(c) \cdot \mathsf{load}(c) \cdot \mathsf{commit}(c) \\
\cdot \quad & \mathsf{fetch}(d) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d) \\
\cdot \quad & \mathsf{fetch}(a) \cdot \mathsf{commit}(a) \cdot \mathsf{prop}(a, 1) \cdot \mathsf{prop}(a, 2)
\end{aligned}
$$

Normal-form computation:

$$
\begin{aligned}
\tau'' \quad = \quad & (\mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a)) \cdot \mathsf{fetch}(b) \\
\cdot \quad & (\mathsf{commit}(a) \cdot \mathsf{prop}(a, 1)) \cdot \mathsf{commit}(b) \cdot \mathsf{prop}(b, 1) \cdot \mathsf{prop}(b, 2) \\
\cdot \quad & (\mathsf{load}(c) \cdot \mathsf{commit}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d))
\end{aligned}
$$

Normal-Form Computations 4/4

Example

A shortest computation with cyclic happens-before relation:

$$\begin{aligned}
\tau \quad &= \quad (\mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a)) \cdot \cancel{\mathsf{fetch}(b)} \\
&\cdot \quad (\mathsf{commit}(a) \cdot \mathsf{prop}(a, 1)) \cdot \cancel{\mathsf{commit}(b)} \cdot \cancel{\mathsf{prop}(b, 1)} \cdot \cancel{\mathsf{prop}(b, 2)} \\
&\cdot \quad (\mathsf{load}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d) \cdot \mathsf{commit}(c))
\end{aligned}$$

Matching sequentially consistent computation:

$$\begin{aligned}
\sigma \quad &= \quad \mathsf{fetch}(c) \cdot \mathsf{load}(c) \cdot \mathsf{commit}(c) \\
&\cdot \quad \mathsf{fetch}(d) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d) \\
&\cdot \quad \mathsf{fetch}(a) \cdot \mathsf{commit}(a) \cdot \mathsf{prop}(a, 1) \cdot \mathsf{prop}(a, 2)
\end{aligned}$$

Normal-form computation:

$$\begin{aligned}
\tau'' \quad &= \quad (\mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a)) \cdot \mathsf{fetch}(b) \\
&\cdot \quad (\mathsf{commit}(a) \cdot \mathsf{prop}(a, 1)) \cdot \mathsf{commit}(b) \cdot \mathsf{prop}(b, 1) \cdot \mathsf{prop}(b, 2) \\
&\cdot \quad (\mathsf{load}(c) \cdot \mathsf{commit}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d))
\end{aligned}$$

Introduction
Power Architecture
Robustness
Deciding Robustness
Characterization of Violating Computations
Normal-Form Computations
Generating Normal-Form Computations
Checking Cyclicity of Happens-Before Relation
Complexity
Conclusion
Related Work
Summary

Generating Normal-Form Computations 1/2

Challenge

Describe the language $\mathcal{L}$ of all normal-form computations of a given degree.

Generating Normal-Form Computations 1/2

Challenge

Describe the language $\mathcal{L}$ of all normal-form computations of a given degree.

We need a language class that

- includes $\mathcal{L}$,
- is closed under intersection with regular languages ($\mathcal{L} \cap \mathcal{R}$),
- has decidable emptiness problem ($\mathcal{L} \cap \mathcal{R} \stackrel{?}{=} \emptyset$).

Generating Normal-Form Computations 1/2

Challenge

Describe the language $\mathcal{L}$ of all normal-form computations of a given degree.

We need a language class that

- includes $\mathcal{L}$,
- is closed under intersection with regular languages ($\mathcal{L} \cap \mathcal{R}$),
- has decidable emptiness problem ($\mathcal{L} \cap \mathcal{R} \stackrel{?}{=} \emptyset$).

Properties of $\mathcal{L}$

- Number of concurrently executed instructions is unbounded $\Rightarrow$ not regular.

Generating Normal-Form Computations 1/2

Challenge

Describe the language $\mathcal{L}$ of all normal-form computations of a given degree.

We need a language class that

- includes $\mathcal{L}$,
- is closed under intersection with regular languages ($\mathcal{L} \cap \mathcal{R}$),
- has decidable emptiness problem ($\mathcal{L} \cap \mathcal{R} \stackrel{?}{=} \emptyset$).

Properties of $\mathcal{L}$

- Number of concurrently executed instructions is unbounded $\Rightarrow$ not regular.
- Can include computations like $(\text{fetch})^n \cdot (\text{load})^n \cdot (\text{commit})^n$ $\Rightarrow$ not even context-free.

Generating Normal-Form computations 2/2

Solution

Define $\mathcal{L}$ as a language of a multiheaded automaton.

Generating Normal-Form computations 2/2

Solution

Define $\mathcal{L}$ as a language of a multiheaded automaton.

Definition (Multiheaded Automaton)

An *n*-headed automaton is an extension of NFA generating n parts of a computation simultaneously, one by each head.

Generating Normal-Form computations 2/2

Solution

Define $\mathcal{L}$ as a language of a multiheaded automaton.

Definition (Multiheaded Automaton)

An *n*-headed automaton is an extension of NFA generating *n* parts of a computation simultaneously, one by each head.

Example (Generating τ'' with a 3-headed Automaton)

$$\tau'' = \\ . \\ .$$

Generating Normal-Form computations 2/2

Solution

Define $\mathcal{L}$ as a language of a multiheaded automaton.

Definition (Multiheaded Automaton)

An *n*-headed automaton is an extension of NFA generating n parts of a computation simultaneously, one by each head.

Example (Generating τ'' with a 3-headed Automaton)

$$\tau'' = \mathsf{fetch}(c)$$

.

.

Generating Normal-Form computations 2/2

Solution

Define $\mathcal{L}$ as a language of a multiheaded automaton.

Definition (Multiheaded Automaton)

An *n*-headed automaton is an extension of NFA generating n parts of a computation simultaneously, one by each head.

Example (Generating τ'' with a 3-headed Automaton)

$$\begin{aligned} \tau'' &= \mathsf{fetch}(c) \\ &\cdot \\ &\cdot \ \mathsf{load}(c) \end{aligned}$$

Generating Normal-Form computations 2/2

Solution

Define $\mathcal{L}$ as a language of a multiheaded automaton.

Definition (Multiheaded Automaton)

An *n*-headed automaton is an extension of NFA generating n parts of a computation simultaneously, one by each head.

Example (Generating τ'' with a 3-headed Automaton)

$$\begin{aligned}\tau'' \quad &= \quad \mathsf{fetch}(c) \\ &\cdot \\ &\cdot \quad \mathsf{load}(c) \cdot \mathsf{commit}(c)\end{aligned}$$

Generating Normal-Form computations 2/2

Solution

Define $\mathcal{L}$ as a language of a multiheaded automaton.

Definition (Multiheaded Automaton)

An *n*-headed automaton is an extension of NFA generating n parts of a computation simultaneously, one by each head.

Example (Generating τ'' with a 3-headed Automaton)

$$
\begin{aligned}
\tau'' &= \mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \\
&\cdot \\
&\cdot \quad \mathsf{load}(c) \cdot \mathsf{commit}(c)
\end{aligned}
$$

Generating Normal-Form computations 2/2

Solution

Define $\mathcal{L}$ as a language of a multiheaded automaton.

Definition (Multiheaded Automaton)

An *n*-headed automaton is an extension of NFA generating n parts of a computation simultaneously, one by each head.

Example (Generating τ'' with a 3-headed Automaton)

$$\begin{aligned}\tau'' &= \mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \\ &\cdot \\ &\cdot \quad \mathsf{load}(c) \cdot \mathsf{commit}(c) \cdot \mathsf{load}(d)\end{aligned}$$

Generating Normal-Form computations 2/2

Solution

Define $\mathcal{L}$ as a language of a multiheaded automaton.

Definition (Multiheaded Automaton)

An *n*-headed automaton is an extension of NFA generating *n* parts of a computation simultaneously, one by each head.

Example (Generating τ'' with a 3-headed Automaton)

$$
\begin{aligned}
\tau'' \quad &= \quad \mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \\
&\cdot \\
&\cdot \quad \mathsf{load}(c) \cdot \mathsf{commit}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d)
\end{aligned}
$$

Generating Normal-Form computations 2/2

Solution

Define $\mathcal{L}$ as a language of a multiheaded automaton.

Definition (Multiheaded Automaton)

An *n*-headed automaton is an extension of NFA generating n parts of a computation simultaneously, one by each head.

Example (Generating τ'' with a 3-headed Automaton)

$$\begin{aligned}\tau'' &= \mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a)\\ &\cdot \\ &\cdot \ \mathsf{load}(c) \cdot \mathsf{commit}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d)\end{aligned}$$

Generating Normal-Form computations 2/2

Solution

Define $\mathcal{L}$ as a language of a multiheaded automaton.

Definition (Multiheaded Automaton)

An *n*-headed automaton is an extension of NFA generating *n* parts of a computation simultaneously, one by each head.

Example (Generating τ'' with a 3-headed Automaton)

$$\begin{aligned}\tau'' \quad = \quad & \mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a) \\ \cdot \quad & \mathsf{commit}(a) \\ \cdot \quad & \mathsf{load}(c) \cdot \mathsf{commit}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d)\end{aligned}$$

Generating Normal-Form computations 2/2

Solution

Define $\mathcal{L}$ as a language of a multiheaded automaton.

Definition (Multiheaded Automaton)

An *n*-headed automaton is an extension of NFA generating *n* parts of a computation simultaneously, one by each head.

Example (Generating τ'' with a 3-headed Automaton)

$$
\begin{aligned}
\tau'' \quad = \quad & \mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a) \\
\cdot \quad & \mathsf{commit}(a) \cdot \mathsf{prop}(a, 1) \\
\cdot \quad & \mathsf{load}(c) \cdot \mathsf{commit}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d)
\end{aligned}
$$

Generating Normal-Form computations 2/2

Solution

Define $\mathcal{L}$ as a language of a multiheaded automaton.

Definition (Multiheaded Automaton)

An *n*-headed automaton is an extension of NFA generating *n* parts of a computation simultaneously, one by each head.

Example (Generating τ'' with a 3-headed Automaton)

$$\begin{array}{rcl}\tau'' & = & \mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a) \cdot \mathsf{fetch}(b) \\ & \cdot & \mathsf{commit}(a) \cdot \mathsf{prop}(a, 1) \\ & \cdot & \mathsf{load}(c) \cdot \mathsf{commit}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d)\end{array}$$

Generating Normal-Form computations 2/2

Solution

Define $\mathcal{L}$ as a language of a multiheaded automaton.

Definition (Multiheaded Automaton)

An *n*-headed automaton is an extension of NFA generating *n* parts of a computation simultaneously, one by each head.

Example (Generating τ'' with a 3-headed Automaton)

$$
\begin{aligned}
\tau'' \quad = \quad & \mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a) \cdot \mathsf{fetch}(b) \\
\cdot \quad & \mathsf{commit}(a) \cdot \mathsf{prop}(a, 1) \cdot \mathsf{commit}(b) \\
\cdot \quad & \mathsf{load}(c) \cdot \mathsf{commit}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d)
\end{aligned}
$$

Generating Normal-Form computations 2/2

Solution

Define $\mathcal{L}$ as a language of a multiheaded automaton.

Definition (Multiheaded Automaton)

An *n*-headed automaton is an extension of NFA generating n parts of a computation simultaneously, one by each head.

Example (Generating τ'' with a 3-headed Automaton)

$$
\begin{aligned}
\tau'' \quad = \quad & \mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a) \cdot \mathsf{fetch}(b) \\
\cdot \quad & \mathsf{commit}(a) \cdot \mathsf{prop}(a, 1) \cdot \mathsf{commit}(b) \cdot \mathsf{prop}(b, 1) \\
\cdot \quad & \mathsf{load}(c) \cdot \mathsf{commit}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d)
\end{aligned}
$$

Generating Normal-Form computations 2/2

Solution

Define $\mathcal{L}$ as a language of a multiheaded automaton.

Definition (Multiheaded Automaton)

An *n*-headed automaton is an extension of NFA generating n parts of a computation simultaneously, one by each head.

Example (Generating τ'' with a 3-headed Automaton)

$$
\begin{aligned}
\tau'' \quad = \quad & \mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a) \cdot \mathsf{fetch}(b) \\
\cdot \quad & \mathsf{commit}(a) \cdot \mathsf{prop}(a, 1) \cdot \mathsf{commit}(b) \cdot \mathsf{prop}(b, 1) \cdot \mathsf{prop}(b, 2) \\
\cdot \quad & \mathsf{load}(c) \cdot \mathsf{commit}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d)
\end{aligned}
$$

Introduction

Power Architecture

Robustness

Deciding Robustness

Characterization of Violating Computations

Normal-Form Computations

Generating Normal-Form Computations

Checking Cyclicity of Happens-Before Relation

Complexity

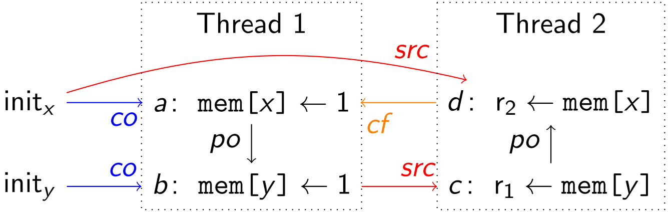
Conclusion

Related Work

Summary

Checking Cyclicity of Happens-Before Relation

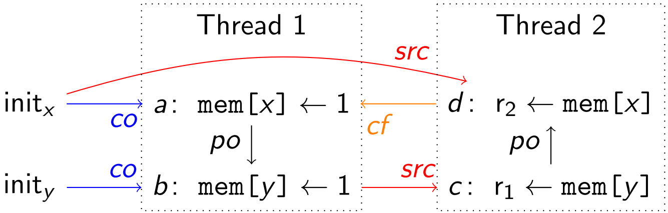
Example (Happens-Before Relation of τ'')



Solution

Checking Cyclicity of Happens-Before Relation

Example (Happens-Before Relation of τ'')

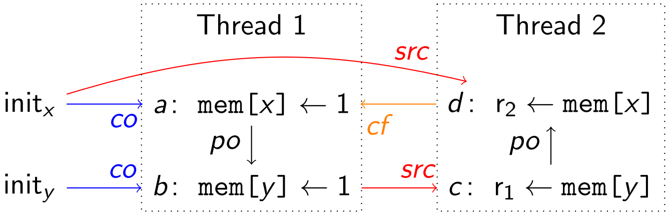


Solution

- The multiheaded automaton in each thread picks two instructions in program order.

Checking Cyclicity of Happens-Before Relation

Example (Happens-Before Relation of τ'')



Solution

- The multiheaded automaton in each thread picks two instructions in program order.
- Finite automata check edges between picked instructions from different threads.

Introduction
- Power Architecture
- Robustness

Deciding Robustness
- Characterization of Violating Computations
- Normal-Form Computations
- Generating Normal-Form Computations
- Checking Cyclicity of Happens-Before Relation
- Complexity

Conclusion
- Related Work
- Summary

Complexity

Theorem

Assuming finite memory, robustness is PSPACE-*complete.*

Proof.

Complexity

Theorem

Assuming finite memory, robustness is $\textsc{PSpace}$*-complete.*

Proof.

- Upper bound: $\mathcal{L} \cap \mathcal{R} \stackrel{?}{=} \emptyset$.

Complexity

Theorem

Assuming finite memory, robustness is $\mathrm{PS{\scriptsize PACE}}$*-complete.*

Proof.

- Upper bound: $\mathcal{L} \cap \mathcal{R} \stackrel{?}{=} \emptyset$.
- Lower bound: SC state reachability [Kozen, 1977].

$\square$

Introduction

- Power Architecture
- Robustness

Deciding Robustness

- Characterization of Violating Computations
- Normal-Form Computations
- Generating Normal-Form Computations
- Checking Cyclicity of Happens-Before Relation
- Complexity

Conclusion

- Related Work
- Summary

Related Work

Related Work

- Robustness
  - Characterization: [Shasha and Snir, 1988].
  - Monitoring algorithms:
    - [Burckhardt and Musuvathi, 2008] (TSO-only, broken),
    - [Burnim et al., 2011] (TSO, PSO).
  - Static overapproximation and fence insertion:
    [Alglave and Maranget, 2011] (TSO, Power).
  - Decidability
    - [Bouajjani et al., 2011] (TSO, PSPACE-completeness),
    - [Bouajjani et al., 2013] (TSO, reduction to SC reachability, fence insertion).

Related Work

- Robustness
  - Characterization: [Shasha and Snir, 1988].
  - Monitoring algorithms:
    - [Burckhardt and Musuvathi, 2008] (TSO-only, broken),
    - [Burnim et al., 2011] (TSO, PSO).
  - Static overapproximation and fence insertion: [Alglave and Maranget, 2011] (TSO, Power).
  - Decidability
    - [Bouajjani et al., 2011] (TSO, PSPACE-completeness),
    - [Bouajjani et al., 2013] (TSO, reduction to SC reachability, fence insertion).
- State reachability
  - PSPACE for SC [Kozen, 1977],
  - Non-primitive recursive for TSO [Atig et al., 2010].

Related Work

- Robustness
  - Characterization: [Shasha and Snir, 1988].
  - Monitoring algorithms:
    - [Burckhardt and Musuvathi, 2008] (TSO-only, broken),
    - [Burnim et al., 2011] (TSO, PSO).
  - Static overapproximation and fence insertion:
    [Alglave and Maranget, 2011] (TSO, Power).
  - Decidability
    - [Bouajjani et al., 2011] (TSO, PSPACE-completeness),
    - [Bouajjani et al., 2013] (TSO, reduction to SC reachability, fence insertion).
- State reachability
  - PSPACE for SC [Kozen, 1977],
  - Non-primitive recursive for TSO [Atig et al., 2010].
- Power models:
  - [Sarkar et al., 2011] (operational),
  - [Mador-Haim et al., 2012] (axiomatic),
  - [Alglave et al., 2013] (overview, newer axiomatic),
  - [Maranget et al.,] (tutorial, with ARM).

Summary

Reduction of Robustness to Language Emptiness

- Look only for normal-form violating computations.
- Use multiheaded automata to generate normal-form computations.
- Check cyclicity of happens-before by regular intersection.

Summary

Reduction of Robustness to Language Emptiness

- Look only for normal-form violating computations.
- Use multiheaded automata to generate normal-form computations.
- Check cyclicity of happens-before by regular intersection.

Robustness against Power is PSPACE-complete

- Upper bound: reduction to language emptiness.
- Lower bound: sequentially consistent state reachability.

Summary

Reduction of Robustness to Language Emptiness

- Look only for normal-form violating computations.
- Use multiheaded automata to generate normal-form computations.
- Check cyclicity of happens-before by regular intersection.

Robustness against Power is PSPACE-complete

- Upper bound: reduction to language emptiness.
- Lower bound: sequentially consistent state reachability.

First decidability result for Power!

Summary

Reduction of Robustness to Language Emptiness

- Look only for normal-form violating computations.
- Use multiheaded automata to generate normal-form computations.
- Check cyclicity of happens-before by regular intersection.

Robustness against Power is PSPACE-complete

- Upper bound: reduction to language emptiness.
- Lower bound: sequentially consistent state reachability.

First decidability result for Power!

Thank you for your attention.
Questions?
derevenetc@cs.uni-kl.de

References I

Alglave, J. and Maranget, L. (2011).
Stability in weak memory models.
In *CAV*, volume 6806 of *LNCS*, pages 50–66. Springer.

Alglave, J., Maranget, L., and Tautschnig, M. (2013).
Herding cats.
CoRR, abs/1308.6810.

Atig, M. F., Bouajjani, A., Burckhardt, S., and Musuvathi, M. (2010).
On the verification problem for weak memory models.
In *POPL*, pages 7–18. ACM.

Bouajjani, A., Derevenetc, E., and Meyer, R. (2013).
Checking and enforcing robustness against TSO.
In *ESOP*, LNCS, pages 533–553. Springer.

References II

Bouajjani, A., Meyer, R., and Möhlmann, E. (2011).
Deciding robustness against Total Store Ordering.
In *ICALP*, volume 6756 of *LNCS*, pages 428–440. Springer.

Burckhardt, S. and Musuvathi, M. (2008).
Effective program verification for relaxed memory models.
In *CAV*, volume 5123 of *LNCS*, pages 107–120. Springer.

Burnim, J., Stergiou, C., and Sen, K. (2011).
Sound and complete monitoring of sequential consistency for relaxed memory models.
In *TACAS*, volume 6605 of *LNCS*, pages 11–25. Springer.

Kozen, D. (1977).
Lower bounds for natural proof systems.
In *FOCS*, pages 254–266. IEEE.

References III

Lamport, L. (1979).
How to make a multiprocessor computer that correctly executes multiprocess programs.
IEEE Transactions on Computers, 28(9):690–691.

Mador-Haim, S., Maranget, L., Sarkar, S., Memarian, K., Alglave, J., Owens, S., Alur, R., Martin, M. M. K., Sewell, P., and Williams, D. (2012).
An axiomatic memory model for power multiprocessors.
In *CAV*, pages 495–512. Springer.

Maranget, L., Sarkar, S., and Sewell, P.
A tutorial introduction to the ARM and POWER relaxed memory models.
https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf.
Draft.

References IV

Sarkar, S., Sewell, P., Alglave, J., Maranget, L., and Williams, D. (2011).
Understanding POWER multiprocessors.
In *PLDI*, pages 175–186. ACM.

Shasha, D. and Snir, M. (1988).
Efficient and correct execution of parallel programs that share memory.
ACM TOPLAS, 10(2):282–312.