Modeling systems via register machines

for the verification of weak memory models

Elli Anastasiadi [1,2] Samuel Grahn [1]

[1]Department of Information Technology, Uppsala University, Sweden
firstname.lastname@it.uu.se

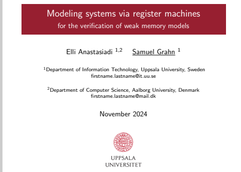
[2]Department of Computer Science, Aalborg University, Denmark
firstname.lastname@mail.dk

November 2024

UPPSALA
UNIVERSITET

2024-10-30

Modeling via Register Machines


Modeling systems via register machines
for the verification of weak memory models

Elli Anastasiadi [1,2] Samuel Grahn [1]

[1]Department of Information Technology, Uppsala University, Sweden
firstname.lastname@it.uu.se

[2]Department of Computer Science, Aalborg University, Denmark
firstname.lastname@mail.dk

November 2024

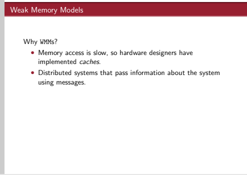
UPPSALA
UNIVERSITET

Introduce Elli and Samuel, title

Weak Memory Models

Why `WMMs`?

- Memory access is slow, so hardware designers have implemented *caches*.
- Distributed systems that pass information about the system using messages.


Weak Memory Models

Why WMMs?

- Memory access is slow, so hardware designers have implemented *caches*.
- Distributed systems that pass information about the system using messages.

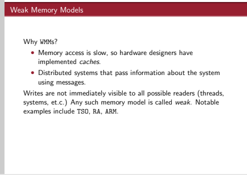
Two situations (next)
The question we want to answer: (next)
(verification problem)
What do we do with undecidability? (next)
Simplify to find a case that is decidable

Weak Memory Models

Why WMMs?

- Memory access is slow, so hardware designers have implemented *caches*.
- Distributed systems that pass information about the system using messages.

Writes are not immediately visible to all possible readers (threads, systems, et.c.) Any such memory model is called *weak*. Notable examples include TSO, RA, ARM.


Weak Memory Models

Why WMMs?
- Memory access is slow, so hardware designers have implemented *caches*.
- Distributed systems that pass information about the system using messages.

Writes are not immediately visible to all possible readers (threads, systems, et.c.) Any such memory model is called *weak*. Notable examples include TSO, RA, ARM.

Two situations (next)
The question we want to answer: (next)
(verification problem)
What do we do with undecidability? (next)
Simplify to find a case that is decidable

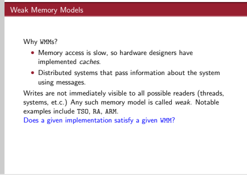
Weak Memory Models

Why `WMMs`?

- Memory access is slow, so hardware designers have implemented *caches*.
- Distributed systems that pass information about the system using messages.

Writes are not immediately visible to all possible readers (threads, systems, et.c.) Any such memory model is called *weak*. Notable examples include `TSO`, `RA`, `ARM`.

Does a given implementation satisfy a given `WMM`?


Weak Memory Models
Why WMMs?
• Memory access is slow, so hardware designers have implemented *caches*.
• Distributed systems that pass information about the system using messages.
Writes are not immediately visible to all possible readers (threads, systems, et.c.) Any such memory model is called *weak*. Notable examples include TSO, RA, ARM.
Does a given implementation satisfy a given WMM?

Two situations (next)
The question we want to answer: (next) (verification problem)
What do we do with undecidability? (next)
Simplify to find a case that is decidable

Weak Memory Models

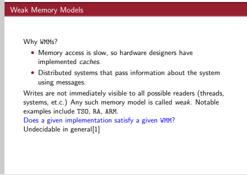
Why WMMs?

- Memory access is slow, so hardware designers have implemented *caches*.
- Distributed systems that pass information about the system using messages.

Writes are not immediately visible to all possible readers (threads, systems, et.c.) Any such memory model is called *weak*. Notable examples include TSO, RA, ARM.

Does a given implementation satisfy a given WMM?

Undecidable in general[1]



Two situations (next)
The question we want to answer: (next)
(verification problem)
What do we do with undecidability? (next)
Simplify to find a case that is decidable

Weak Memory Models

Why WMMs?

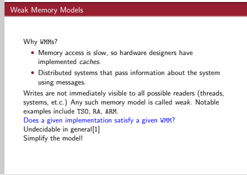
- Memory access is slow, so hardware designers have implemented *caches*.
- Distributed systems that pass information about the system using messages.

Writes are not immediately visible to all possible readers (threads, systems, et.c.) Any such memory model is called *weak*. Notable examples include TSO, RA, ARM.

Does a given implementation satisfy a given WMM?

Undecidable in general[1]

Simplify the model!


Weak Memory Models

Why WMMs?

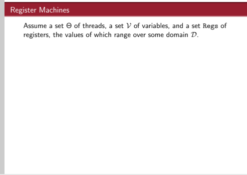
- Memory access is slow, so hardware designers have implemented *caches*.
- Distributed systems that pass information about the system using messages.

Writes are not immediately visible to all possible readers (threads, systems, et.c.) Any such memory model is called *weak*. Notable examples include TSO, RA, ARM.
Does a given implementation satisfy a given WMM?
Undecidable in general[1]
Simplify the model!

Two situations (next)
The question we want to answer: (next)
(verification problem)
What do we do with undecidability? (next)
Simplify to find a case that is decidable

Register Machines

Assume a set Θ of threads, a set $\mathcal{V}$ of variables, and a set $\mathtt{Regs}$ of registers, the values of which range over some domain $\mathcal{D}$.



Register Machines

Assume a set Θ of threads, a set $\mathcal{V}$ of variables, and a set $\mathtt{Regs}$ of registers, the values of which range over some domain $\mathcal{D}$.

Formal definition.

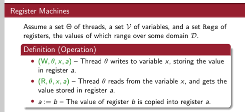
Intuitively: FSA with operation labels

Register Machines

Assume a set Θ of threads, a set $\mathcal{V}$ of variables, and a set $\mathtt{Regs}$ of registers, the values of which range over some domain $\mathcal{D}$.

Definition (Operation)

- $(\mathrm{W}, \theta, x, a)$ – Thread θ writes to variable x, storing the value in register a.
- $(\mathrm{R}, \theta, x, a)$ – Thread θ reads from the variable x, and gets the value stored in register a.
- $a := b$ – The value of register b is copied into register a.


Register Machines

Assume a set Θ of threads, a set $\mathcal{V}$ of variables, and a set $\mathtt{Regs}$ of registers, the values of which range over some domain $\mathcal{D}$.

Definition (Operation)

- $(\mathrm{W}, \theta, x, a)$ – Thread θ writes to variable x, storing the value in register a.
- $(\mathrm{R}, \theta, x, a)$ – Thread θ reads from the variable x, and gets the value stored in register a.
- $a := b$ – The value of register b is copied into register a.

Formal definition.

Intuitively: FSA with operation labels

Register Machines

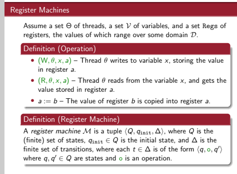
Assume a set Θ of threads, a set $\mathcal{V}$ of variables, and a set $\mathtt{Regs}$ of registers, the values of which range over some domain $\mathcal{D}$.

Definition (Operation)

- $(\mathsf{W}, \theta, x, a)$ – Thread θ writes to variable x, storing the value in register a.
- $(\mathsf{R}, \theta, x, a)$ – Thread θ reads from the variable x, and gets the value stored in register a.
- $a := b$ – The value of register b is copied into register a.

Definition (Register Machine)

A *register machine* $\mathcal{M}$ is a tuple $\langle Q, q_{\mathtt{init}}, \Delta\rangle$, where Q is the (finite) set of states, $q_{\mathtt{init}} \in Q$ is the initial state, and Δ is the finite set of transitions, where each $t \in \Delta$ is of the form $\langle q, \mathsf{o}, q'\rangle$ where $q, q' \in Q$ are states and o is an operation.


Register Machines

Assume a set Θ of threads, a set $\mathcal{V}$ of variables, and a set $\mathtt{Regs}$ of registers, the values of which range over some domain $\mathcal{D}$.

Definition (Operation)

- $(\mathsf{W}, \theta, x, a)$ – Thread θ writes to variable x, storing the value in register a.
- $(\mathsf{R}, \theta, x, a)$ – Thread θ reads from the variable x, and gets the value stored in register a.
- $a := b$ – The value of register b is copied into register a.

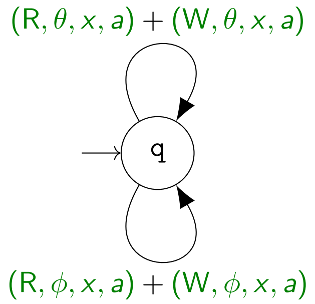
Definition (Register Machine)

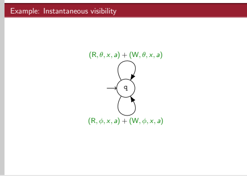
A *register machine* $\mathcal{M}$ is a tuple $\langle Q, q_{\mathtt{init}}, \Delta\rangle$, where Q is the (finite) set of states, $q_{\mathtt{init}} \in Q$ is the initial state, and Δ is the finite set of transitions, where each $t \in \Delta$ is of the form $\langle q, \mathsf{o}, q'\rangle$ where $q, q' \in Q$ are states and o is an operation.

Formal definition.

Intuitively: FSA with operation labels

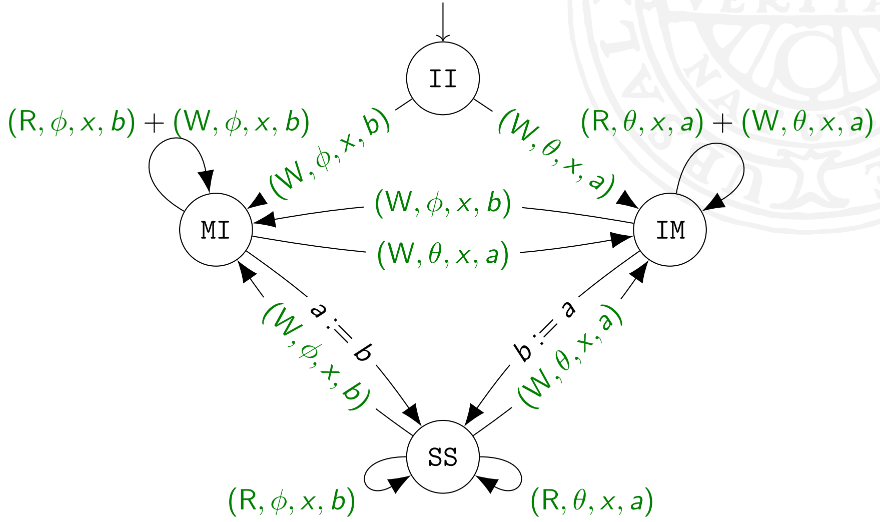
Example: Instantaneous visibility

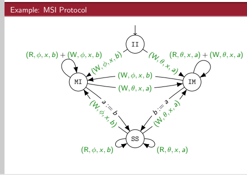

$$\begin{tikzcd} \arrow[r] & q \arrow[loop above, "(\mathrm{R},\theta,x,a)+(\mathrm{W},\theta,x,a)"] \arrow[loop below, "(\mathrm{R},\phi,x,a)+(\mathrm{W},\phi,x,a)"] \end{tikzcd}$$



Example: Single state, two threads that can read and write from the same register

Example: MSI Protocol

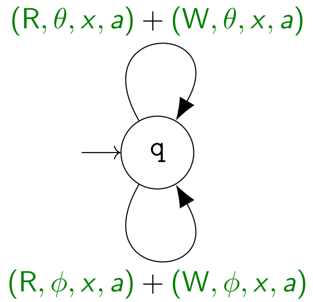

II
MI
IM
SS
$(\mathsf{R}, \phi, x, b) + (\mathsf{W}, \phi, x, b)$
(W, ϕ, x, b)
$(\mathsf{W}, \theta, x, a)$
$(\mathsf{R}, \theta, x, a) + (\mathsf{W}, \theta, x, a)$
(W, ϕ, x, b)
$(\mathsf{W}, \theta, x, a)$
$a := b$
(W, ϕ, x, b)
$b := a$
$(\mathsf{W}, \theta, x, a)$
(R, ϕ, x, b)
$(\mathsf{R}, \theta, x, a)$

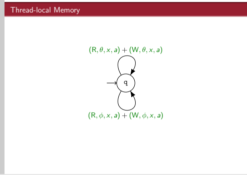


Each thread is in one of three states:
Invalid – Variable not in cache
Modified – Variable owned and controlled
Shared – Read access, shared with others

Modeling via Register Machines
└─Modeling

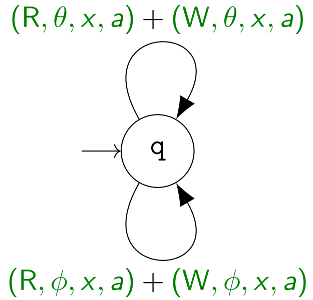
Thread-local Memory


$(\mathrm{R}, \theta, x, a) + (\mathrm{W}, \theta, x, a)$
q
$(\mathrm{R}, \phi, x, a) + (\mathrm{W}, \phi, x, a)$

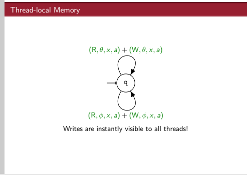

Thread-local Memory

Previous example: single-state read/write In a real system: effects not immediately visible (e.g. cache)

Thread-local Memory

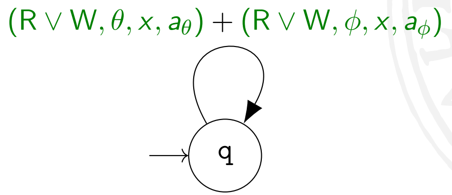

$(\mathsf{R}, \theta, x, a) + (\mathsf{W}, \theta, x, a)$
q
$(\mathsf{R}, \phi, x, a) + (\mathsf{W}, \phi, x, a)$

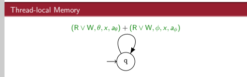
Writes are instantly visible to all threads!



Previous example: single-state read/write In a real system: effects not immediately visible (e.g. cache)

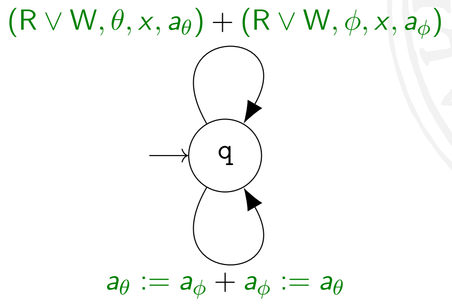
Thread-local Memory

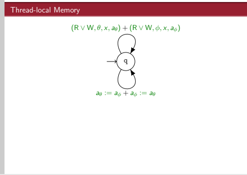

$(\mathsf{R} \vee \mathsf{W}, \theta, x, a_\theta) + (\mathsf{R} \vee \mathsf{W}, \phi, x, a_\phi)$
q



Create thread local registers
(next) Allow copying information between threads
(next) Problem!
(next) Solution!

Thread-local Memory

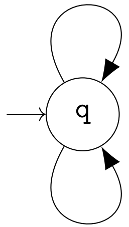

$$\begin{tikzcd} \arrow[r] & q \arrow[loop above, "(\mathsf{R} \vee \mathsf{W}, \theta, x, a_\theta) + (\mathsf{R} \vee \mathsf{W}, \phi, x, a_\phi)"] \arrow[loop below, "a_\theta := a_\phi + a_\phi := a_\theta"] \end{tikzcd}$$



Create thread local registers
(next) Allow copying information between threads
(next) Problem!
(next) Solution!

Thread-local Memory

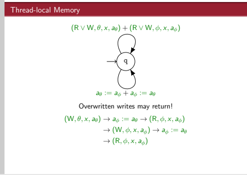
$$(\mathsf{R} \vee \mathsf{W}, \theta, x, a_\theta) + (\mathsf{R} \vee \mathsf{W}, \phi, x, a_\phi)$$



$$a_\theta := a_\phi + a_\phi := a_\theta$$

Overwritten writes may return!

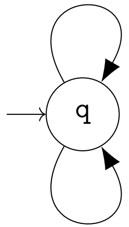
$$\begin{aligned}(\mathsf{W}, \theta, x, a_\theta) &\to a_\phi := a_\theta \to (\mathsf{R}, \phi, x, a_\phi)\\ &\to (\mathsf{W}, \phi, x, a_\phi) \to a_\phi := a_\theta\\ &\to (\mathsf{R}, \phi, x, a_\phi)\end{aligned}$$



Create thread local registers
(next) Allow copying information between threads
(next) Problem!
(next) Solution!

Thread-local Memory

$$(\mathrm{R} \vee \mathrm{W}, \theta, x, a_\theta) + (\mathrm{R} \vee \mathrm{W}, \phi, x, a_\phi)$$

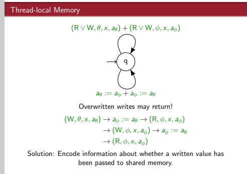


$$a_\theta := a_\phi + a_\phi := a_\theta$$

Overwritten writes may return!

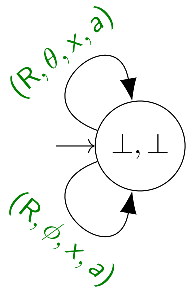
$$\begin{aligned}(\mathrm{W}, \theta, x, a_\theta) &\to a_\phi := a_\theta \to (\mathrm{R}, \phi, x, a_\phi)\\ &\to (\mathrm{W}, \phi, x, a_\phi) \to a_\phi := a_\theta\\ &\to (\mathrm{R}, \phi, x, a_\phi)\end{aligned}$$

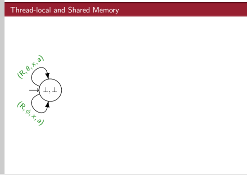
Solution: Encode information about whether a written value has been passed to shared memory.



Create thread local registers
(next) Allow copying information between threads
(next) Problem!
(next) Solution!

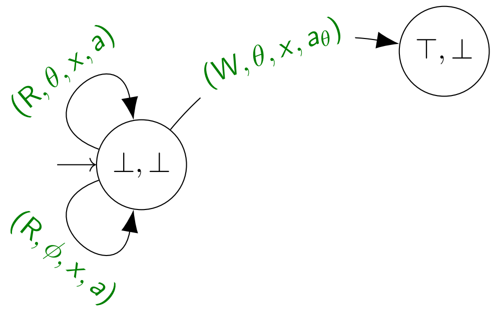
Thread-local and Shared Memory

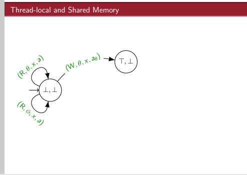




Initial state, none have written. Introduce a, memory
(next) θ writes. (storing in state)
(next) Copy back.
(next) Reads and writes
(next) Symmetricaly for ϕ
(next) From each of the two (end) states, The *other* thread writes.
Both Write and read locally SAY: t is either θ or ϕ

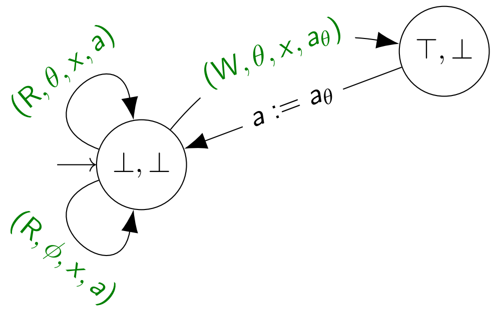
Thread-local and Shared Memory



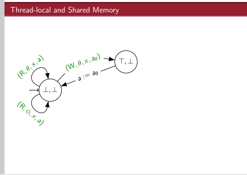


Initial state, none have written. Introduce *a*, memory
(next) θ writes. (storing in state)
(next) Copy back.
(next) Reads and writes
(next) Symmetricaly for ϕ
(next) From each of the two (end) states, The *other* thread writes.
Both Write and read locally SAY: t is either θ or ϕ

Thread-local and Shared Memory

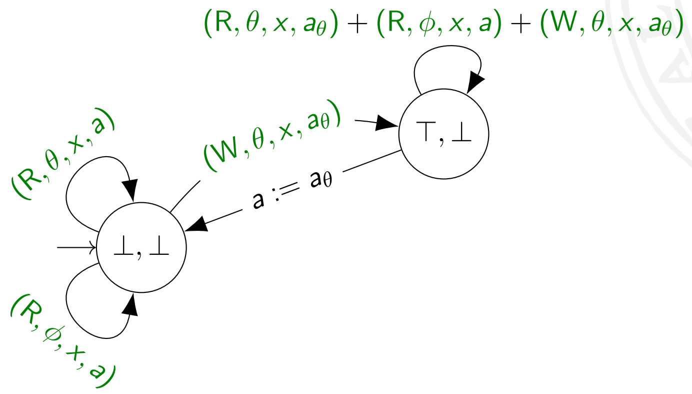


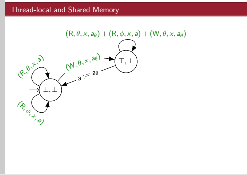
$$\begin{tikzcd}
\to \bot,\bot \arrow[loop above, "{(R,\theta,x,a)}"] \arrow[loop below, "{(R,\phi,x,a)}"] \arrow[r, bend left, "{(W,\theta,x,a_\theta)}"] & \top,\bot \arrow[l, "a := a_\theta"]
\end{tikzcd}$$



Initial state, none have written. Introduce *a*, memory
(next) θ writes. (storing in state)
(next) Copy back.
(next) Reads and writes
(next) Symmetricaly for ϕ
(next) From each of the two (end) states, The *other* thread writes.
Both Write and read locally SAY: t is either θ or ϕ

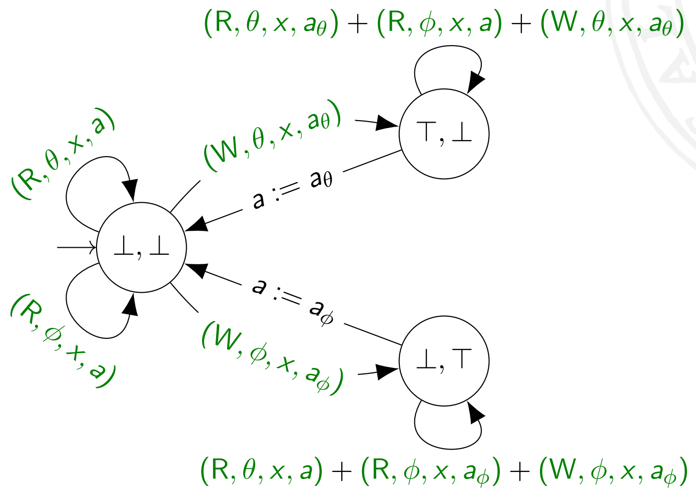
Thread-local and Shared Memory





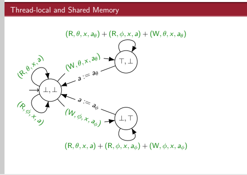
Initial state, none have written. Introduce a, memory
(next) θ writes. (storing in state)
(next) Copy back.
(next) Reads and writes
(next) Symmetricaly for ϕ
(next) From each of the two (end) states, The *other* thread writes.
Both Write and read locally SAY: t is either θ or ϕ

Thread-local and Shared Memory


$(\mathsf{R}, \theta, x, a_\theta) + (\mathsf{R}, \phi, x, a) + (\mathsf{W}, \theta, x, a_\theta)$
$\top, \bot$
$(\mathsf{W}, \theta, x, a_\theta)$
$a := a_\theta$
$(\mathsf{R}, \theta, x, a)$
$\bot, \bot$
(R, ϕ, x, a)
$a := a_\phi$
$(\mathsf{W}, \phi, x, a_\phi)$
$\bot, \top$
$(\mathsf{R}, \theta, x, a) + (\mathsf{R}, \phi, x, a_\phi) + (\mathsf{W}, \phi, x, a_\phi)$

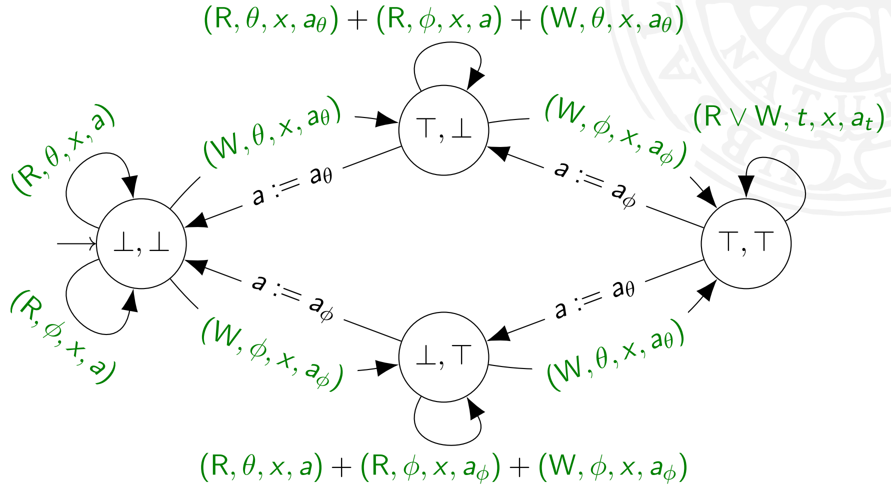
2024-10-30

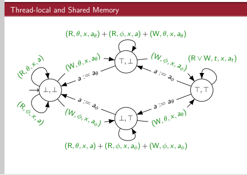
Modeling via Register Machines
└Modeling
└Thread-local and Shared Memory



Initial state, none have written. Introduce *a*, memory
(next) θ writes. (storing in state)
(next) Copy back.
(next) Reads and writes
(next) Symmetricaly for ϕ
(next) From each of the two (end) states, The *other* thread writes.
Both Write and read locally SAY: t is either θ or ϕ

Thread-local and Shared Memory

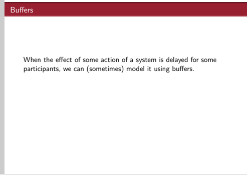

$(\mathsf{R},\theta,x,a_\theta)+(\mathsf{R},\phi,x,a)+(\mathsf{W},\theta,x,a_\theta)$
$\top,\bot$
$(\mathsf{W},\theta,x,a_\theta)$
$(\mathsf{W},\phi,x,a_\phi)$
$(\mathsf{R}\vee\mathsf{W},t,x,a_t)$
(R,θ,x,a)
$a := a_\theta$
$a := a_\phi$
$\bot,\bot$
$\top,\top$
$a := a_\phi$
$a := a_\theta$
(R,ϕ,x,a)
$(\mathsf{W},\phi,x,a_\phi)$
$(\mathsf{W},\theta,x,a_\theta)$
$\bot,\top$
$(\mathsf{R},\theta,x,a)+(\mathsf{R},\phi,x,a_\phi)+(\mathsf{W},\phi,x,a_\phi)$



Initial state, none have written. Introduce a, memory
(next) θ writes. (storing in state)
(next) Copy back.
(next) Reads and writes
(next) Symmetricaly for ϕ
(next) From each of the two (end) states, The *other* thread writes.
Both Write and read locally SAY: t is either θ or ϕ

Buffers

When the effect of some action of a system is delayed for some participants, we can (sometimes) model it using buffers.



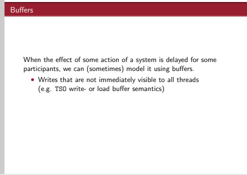
Buffers

When the effect of some action of a system is delayed for some participants, we can (sometimes) model it using buffers.

Buffers

When the effect of some action of a system is delayed for some participants, we can (sometimes) model it using buffers.

- Writes that are not immediately visible to all threads (e.g. `TSO` write- or load buffer semantics)


Buffers

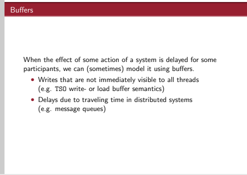
When the effect of some action of a system is delayed for some participants, we can (sometimes) model it using buffers.

- Writes that are not immediately visible to all threads (e.g. TSO write- or load buffer semantics)

Buffers

When the effect of some action of a system is delayed for some participants, we can (sometimes) model it using buffers.

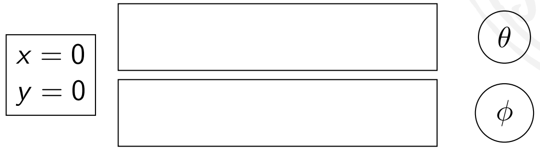
- Writes that are not immediately visible to all threads (e.g. `TSO` write- or load buffer semantics)
- Delays due to traveling time in distributed systems (e.g. message queues)


Buffers

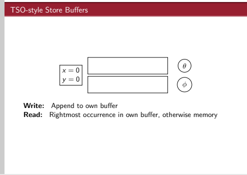
When the effect of some action of a system is delayed for some participants, we can (sometimes) model it using buffers.

- Writes that are not immediately visible to all threads (e.g. TSO write- or load buffer semantics)
- Delays due to traveling time in distributed systems (e.g. message queues)

TSO-style Store Buffers

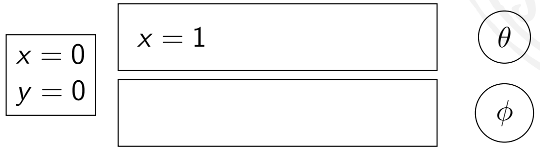


Write: Append to own buffer
Read: Rightmost occurrence in own buffer, otherwise memory

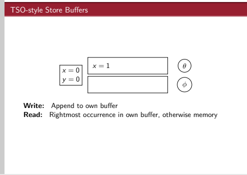


Values in memory, Buffers, one for each thread. DO SOME READS AT EVERY STEP

TSO-style Store Buffers

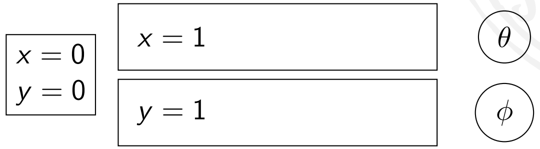


Write: Append to own buffer
Read: Rightmost occurrence in own buffer, otherwise memory

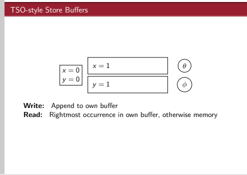


Values in memory, Buffers, one for each thread. DO SOME READS AT EVERY STEP

TSO-style Store Buffers

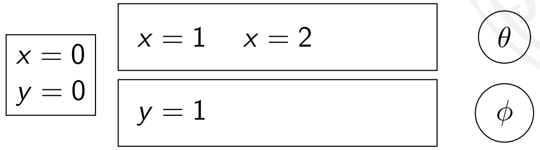


Write: Append to own buffer
Read: Rightmost occurrence in own buffer, otherwise memory

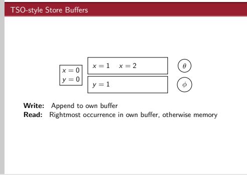


Values in memory, Buffers, one for each thread. DO SOME READS AT EVERY STEP

TSO-style Store Buffers

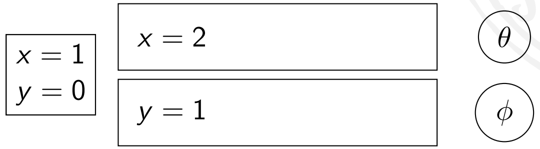


Write: Append to own buffer
Read: Rightmost occurrence in own buffer, otherwise memory

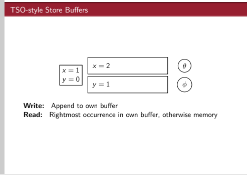


Values in memory, Buffers, one for each thread. DO SOME READS AT EVERY STEP

TSO-style Store Buffers



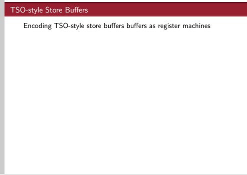
Write: Append to own buffer
Read: Rightmost occurrence in own buffer, otherwise memory



Memory (left) Two threads (right), each with a store buffer (middle)
(Write = append, Read = rightmost)
(next) Say θ writes $x = 1$.
If θ would try to read x, it would read 1. ϕ would read 0.
(next) ϕ writes. (next) θ writes again. The only remaining part is:
(next) Handling buffers: take first message and apply it to memory.

TSO-style Store Buffers

Encoding TSO-style store buffers buffers as register machines

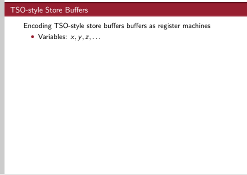

TSO-style Store Buffers

Encoding TSO-style store buffers buffers as register machines

TSO-style Store Buffers

Encoding TSO-style store buffers buffers as register machines

- Variables: $x, y, z, \ldots$


TSO-style Store Buffers

Encoding TSO-style store buffers buffers as register machines

- Variables: $x, y, z, \ldots$

TSO-style Store Buffers

Encoding TSO-style store buffers buffers as register machines

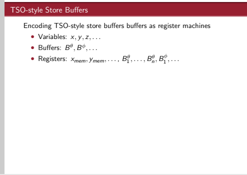
- Variables: $x, y, z, \ldots$
- Buffers: $B^{\theta}, B^{\phi}, \ldots$

TSO-style Store Buffers
Encoding TSO-style store buffers buffers as register machines
- Variables: $x, y, z, \ldots$
- Buffers: $B^{\theta}, B^{\phi}, \ldots$

TSO-style Store Buffers

Encoding TSO-style store buffers buffers as register machines

- Variables: $x, y, z, \ldots$
- Buffers: $B^\theta, B^\phi, \ldots$
- Registers: $x_{mem}, y_{mem}, \ldots,\ B^\theta_1, \ldots, B^\theta_n, B^\phi_1, \ldots$


TSO-style Store Buffers

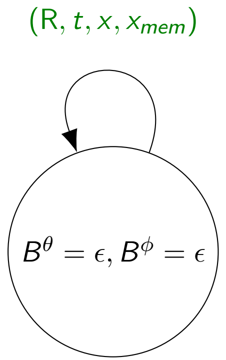
Encoding TSO-style store buffers buffers as register machines

- Variables: $x, y, z, \ldots$
- Buffers: $B^\theta, B^\phi, \ldots$
- Registers: $x_{mem}, y_{mem}, \ldots,\ B^\theta_1, \ldots, B^\theta_n, B^\phi_1, \ldots$

TSO-style Store Buffers

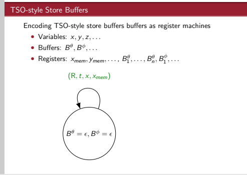
Encoding TSO-style store buffers buffers as register machines

- Variables: $x, y, z, \ldots$
- Buffers: $B^\theta, B^\phi, \ldots$
- Registers: $x_{mem}, y_{mem}, \ldots, B^\theta_1, \ldots, B^\theta_n, B^\phi_1, \ldots$


$(\mathrm{R}, t, x, x_{mem})$
$B^\theta = \epsilon, B^\phi = \epsilon$

2024-10-30

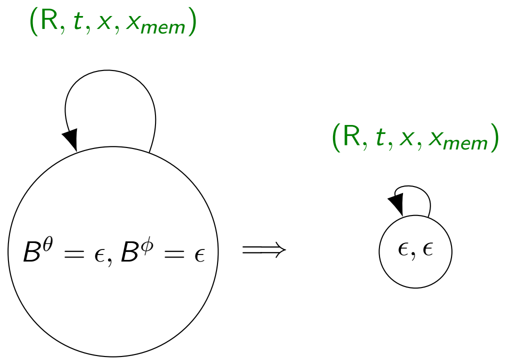
Modeling via Register Machines
└─Modeling
└─TSO-style Store Buffers

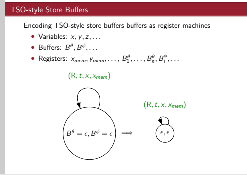


TSO-style Store Buffers

Encoding TSO-style store buffers buffers as register machines

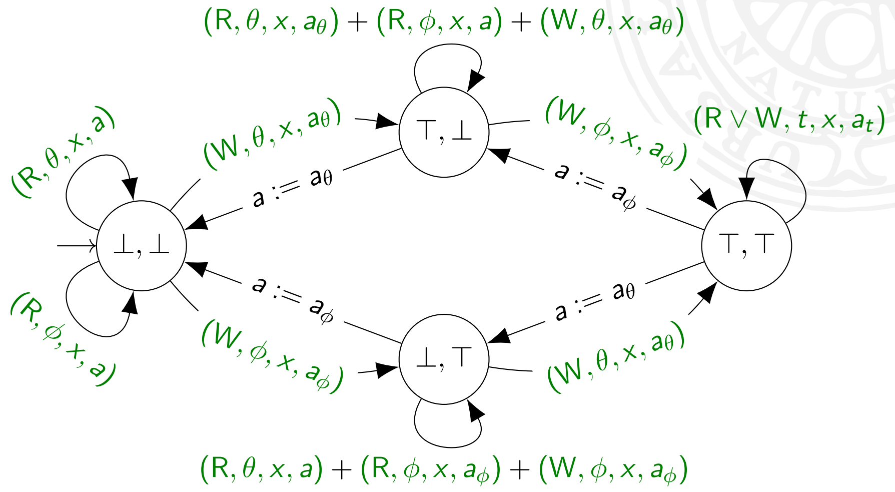
- Variables: $x, y, z, \ldots$
- Buffers: $B^\theta, B^\phi, \ldots$
- Registers: $x_{mem}, y_{mem}, \ldots, B_1^\theta, \ldots, B_n^\theta, B_1^\phi, \ldots$

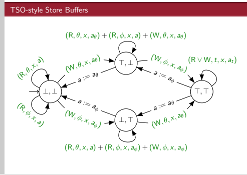

$(\mathrm{R}, t, x, x_{mem})$
$B^\theta = \epsilon, B^\phi = \epsilon$
$\Longrightarrow$
$(\mathrm{R}, t, x, x_{mem})$
ϵ, ϵ



Variables (next)
Buffers = sequence of var names, one for each thread (next)
Registers: one for each variable, and one for each buffer slot (bounded buffers). (next)
We store the buffer contents as part of the state. (next)
But draw it smaller.

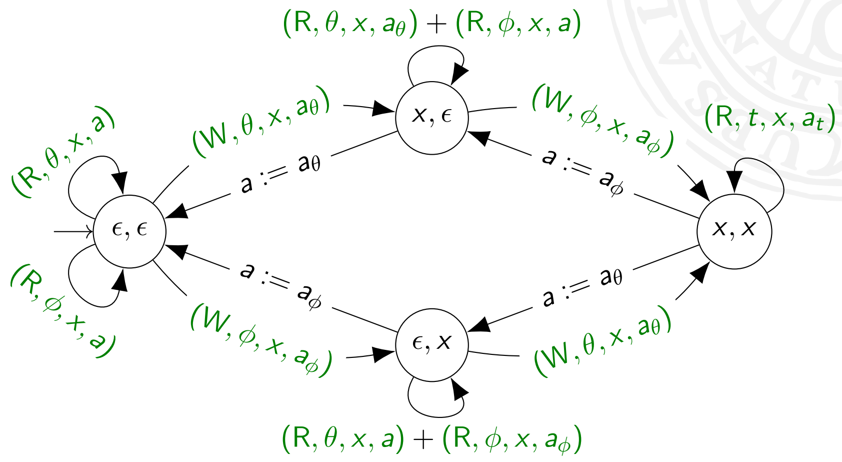
TSO-style Store Buffers

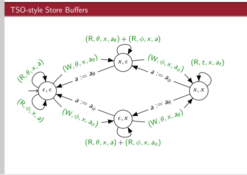

$$\begin{tikzcd}
 & \top,\bot \arrow[loop above, "{(R,\theta,x,a_\theta)+(R,\phi,x,a)+(W,\theta,x,a_\theta)}"] \arrow[dl, "a:=a_\theta"'] \arrow[dr, "{(W,\phi,x,a_\phi)}"] & \\
\bot,\bot \arrow[loop above, "{(R,\theta,x,a)}"] \arrow[loop below, "{(R,\phi,x,a)}"] \arrow[ur, "{(W,\theta,x,a_\theta)}"] \arrow[dr, "{(W,\phi,x,a_\phi)}"'] & & \top,\top \arrow[loop above, "{(R\vee W,t,x,a_t)}"] \arrow[ul, "a:=a_\phi"'] \arrow[dl, "a:=a_\theta"] \\
 & \bot,\top \arrow[loop below, "{(R,\theta,x,a)+(R,\phi,x,a_\phi)+(W,\phi,x,a_\phi)}"] \arrow[ul, "a:=a_\phi"] \arrow[ur, "{(W,\theta,x,a_\theta)}"'] &
\end{tikzcd}$$



We go back and consider the register machine from the context of buffers. Specific instance of the store-buffer: Two threads (i.e. two buffers), one variable, buffer size 1.

TSO-style Store Buffers

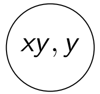


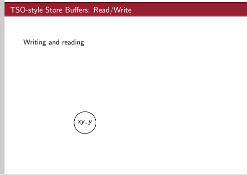


Note, we remove the write loops, as we consider the buffer to be full!
Larger buffers!

TSO-style Store Buffers: Read/Write

Writing and reading





TSO-style Store Buffers: Read/Write

Writing and reading

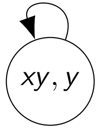
xy, y

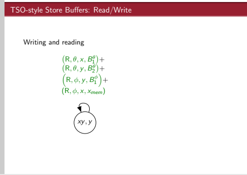
Assume a bound $n > 2$.
From this state, we may read (point out each read) (next)
θ writes (next)
ϕ writes (next)

TSO-style Store Buffers: Read/Write

Writing and reading

$$(\mathsf{R}, \theta, x, B_1^\theta)+$$
$$(\mathsf{R}, \theta, y, B_2^\theta)+$$
$$\left(\mathsf{R}, \phi, y, B_1^\phi\right)+$$
$$(\mathsf{R}, \phi, x, x_{mem})$$





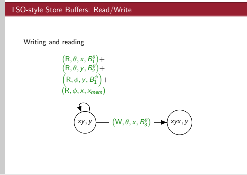
Assume a bound $n > 2$.
From this state, we may read (point out each read) (next)
θ writes (next)
ϕ writes (next)

TSO-style Store Buffers: Read/Write

Writing and reading

$$
\begin{array}{l}
(\mathrm{R}, \theta, x, B_1^{\theta})+ \\
(\mathrm{R}, \theta, y, B_2^{\theta})+ \\
\left(\mathrm{R}, \phi, y, B_1^{\phi}\right)+ \\
(\mathrm{R}, \phi, x, x_{mem})
\end{array}
$$

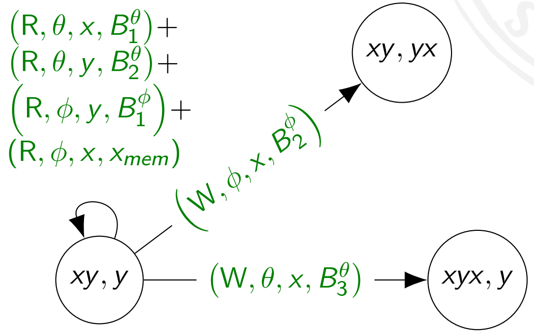
$$\begin{tikzcd}
xy, y \arrow[loop above] \arrow[r, "{(\mathrm{W}, \theta, x, B_3^{\theta})}"] & xyx, y
\end{tikzcd}$$

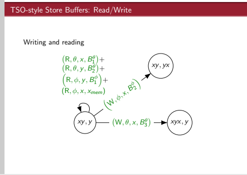

TSO-style Store Buffers: Read/Write
Writing and reading
$(\mathrm{R}, \theta, x, B_1^{\theta})+$ $(\mathrm{R}, \theta, y, B_2^{\theta})+$ $\left(\mathrm{R}, \phi, y, B_1^{\phi}\right)+$ $(\mathrm{R}, \phi, x, x_{mem})$
xy, y $(\mathrm{W}, \theta, x, B_3^{\theta})$ xyx, y

Assume a bound $n > 2$.
From this state, we may read (point out each read) (next)
θ writes (next)
ϕ writes (next)

TSO-style Store Buffers: Read/Write

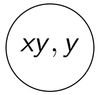
Writing and reading


$(\mathrm{R}, \theta, x, B_1^\theta)+$
$(\mathrm{R}, \theta, y, B_2^\theta)+$
$(\mathrm{R}, \phi, y, B_1^\phi)+$
$(\mathrm{R}, \phi, x, x_{mem})$
xy, yx
$(\mathrm{W}, \phi, x, B_2^\phi)$
xy, y
$(\mathrm{W}, \theta, x, B_3^\theta)$
xyx, y



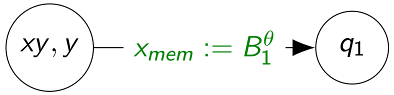
Assume a bound $n > 2$.
From this state, we may read (point out each read) (next)
θ writes (next)
ϕ writes (next)

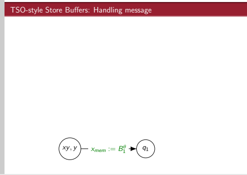
TSO-style Store Buffers: Handling message



Start from the same state.
Handle a message: either from θ or ϕ.
Example: show for θ.
(next) Since first message is x, copy first buffer register to x_{mem}.
DISJOINT sequence of states, not appearing anywhere else! (next) Copy second to first
(next) And so on
(next) Until the final copy
Where we have removed the message from the state.

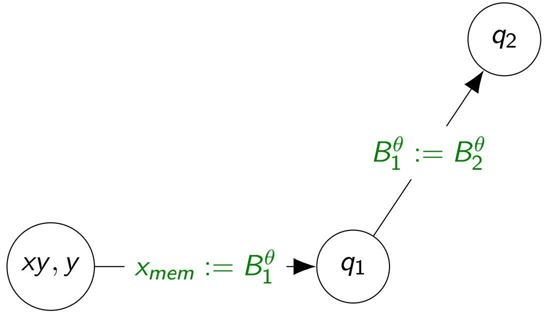
TSO-style Store Buffers: Handling message

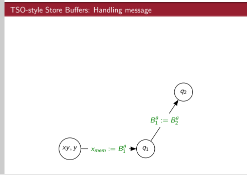




Start from the same state.
Handle a message: either from θ or ϕ.
Example: show for θ.
(next) Since first message is x, copy first buffer register to x_{mem}.
DISJOINT sequence of states, not appearing anywhere else! (next) Copy second to first
(next) And so on
(next) Until the final copy
Where we have removed the message from the state.

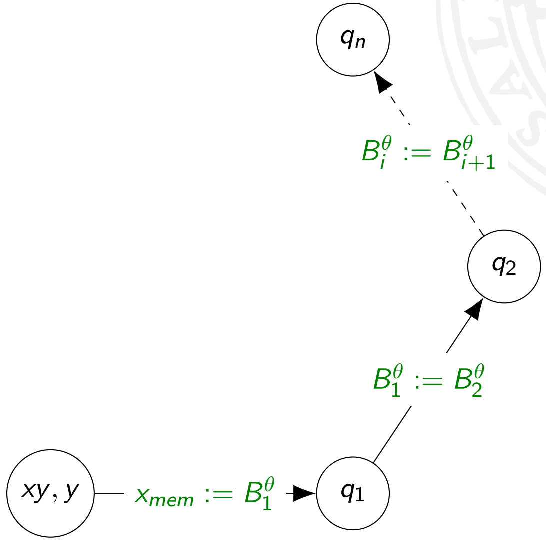
TSO-style Store Buffers: Handling message

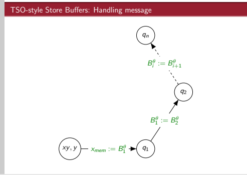




Start from the same state.
Handle a message: either from θ or ϕ.
Example: show for θ.
(next) Since first message is x, copy first buffer register to x_{mem}.
DISJOINT sequence of states, not appearing anywhere else! (next) Copy second to first
(next) And so on
(next) Until the final copy
Where we have removed the message from the state.

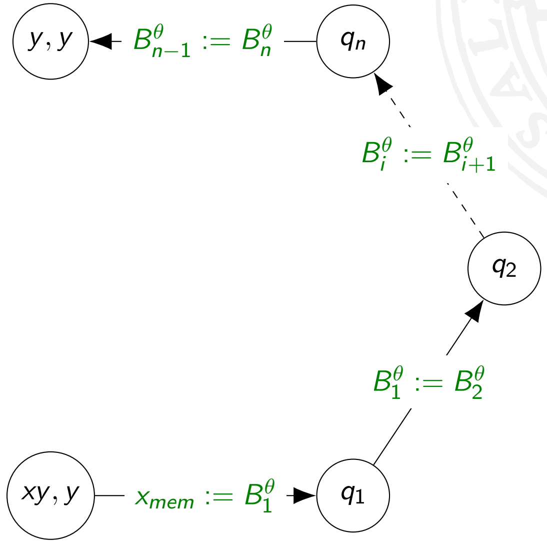
TSO-style Store Buffers: Handling message


$$\begin{tikzcd} & & q_n \\ & & & q_2 \arrow[ul, dashed, "B_i^\theta := B_{i+1}^\theta"] \\ xy, y \arrow[r, "x_{mem} := B_1^\theta"] & & q_1 \arrow[ur, "B_1^\theta := B_2^\theta"] \end{tikzcd}$$

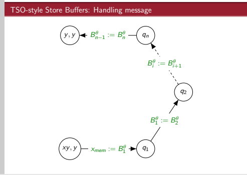


Start from the same state.
Handle a message: either from θ or ϕ.
Example: show for θ.
(next) Since first message is x, copy first buffer register to x_{mem}.
DISJOINT sequence of states, not appearing anywhere else! (next) Copy second to first
(next) And so on
(next) Until the final copy
Where we have removed the message from the state.

TSO-style Store Buffers: Handling message


y, y
$B^{\theta}_{n-1} := B^{\theta}_{n}$
q_n
$B^{\theta}_{i} := B^{\theta}_{i+1}$
q_2
$B^{\theta}_{1} := B^{\theta}_{2}$
xy, y
$x_{mem} := B^{\theta}_{1}$
q_1

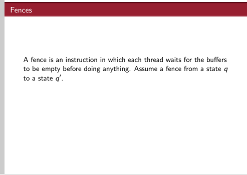
Modeling via Register Machines
└Modeling
└TSO-style Store Buffers: Handling message



Start from the same state.
Handle a message: either from θ or ϕ.
Example: show for θ.
(next) Since first message is x, copy first buffer register to x_{mem}.
DISJOINT sequence of states, not appearing anywhere else! (next) Copy second to first
(next) And so on
(next) Until the final copy
Where we have removed the message from the state.

Fences

A fence is an instruction in which each thread waits for the buffers to be empty before doing anything. Assume a fence from a state q to a state q'.



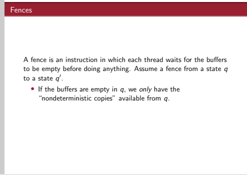
Fences

A fence is an instruction in which each thread waits for the buffers to be empty before doing anything. Assume a fence from a state q to a state q'.

Fences

A fence is an instruction in which each thread waits for the buffers to be empty before doing anything. Assume a fence from a state q to a state q'.

- If the buffers are empty in q, we *only* have the "nondeterministic copies" available from q.


Fences

A fence is an instruction in which each thread waits for the buffers to be empty before doing anything. Assume a fence from a state q to a state q'.

- If the buffers are empty in q, we *only* have the "nondeterministic copies" available from q.

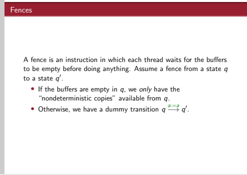
Fences

A fence is an instruction in which each thread waits for the buffers to be empty before doing anything. Assume a fence from a state q to a state q'.

- If the buffers are empty in q, we *only* have the "nondeterministic copies" available from q.
- Otherwise, we have a dummy transition $q \xrightarrow{a:=a} q'$.

2024-10-30

Modeling via Register Machines
└Modeling
└Fences


Fences

A fence is an instruction in which each thread waits for the buffers to be empty before doing anything. Assume a fence from a state q to a state q'.

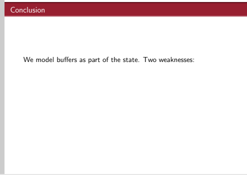
- If the buffers are empty in q, we *only* have the "nondeterministic copies" available from q.
- Otherwise, we have a dummy transition $q \xrightarrow{a:=a} q'$.

2024-10-30

Modeling via Register Machines
└─Conclusion

Conclusion

We model buffers as part of the state. Two weaknesses:

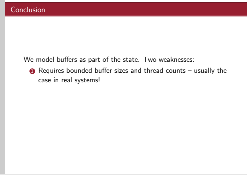

Conclusion

We model buffers as part of the state. Two weaknesses:

Conclusion

We model buffers as part of the state. Two weaknesses:

1. Requires bounded buffer sizes and thread counts – usually the case in real systems!



Conclusion

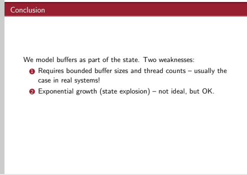
We model buffers as part of the state. Two weaknesses:

1. Requires bounded buffer sizes and thread counts – usually the case in real systems!

Conclusion

We model buffers as part of the state. Two weaknesses:

1. Requires bounded buffer sizes and thread counts – usually the case in real systems!
2. Exponential growth (state explosion) – not ideal, but OK.


Conclusion

We model buffers as part of the state. Two weaknesses:

1. Requires bounded buffer sizes and thread counts – usually the case in real systems!
2. Exponential growth (state explosion) – not ideal, but OK.

Conclusion

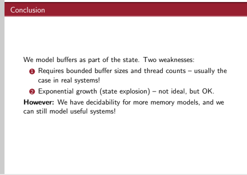
We model buffers as part of the state. Two weaknesses:

1. Requires bounded buffer sizes and thread counts – usually the case in real systems!
2. Exponential growth (state explosion) – not ideal, but OK.

However: We have decidability for more memory models, and we can still model useful systems!

2024-10-30

Modeling via Register Machines
└─Conclusion
  └─Conclusion


Conclusion

We model buffers as part of the state. Two weaknesses:

1. Requires bounded buffer sizes and thread counts – usually the case in real systems!
2. Exponential growth (state explosion) – not ideal, but OK.

However: We have decidability for more memory models, and we can still model useful systems!

References

[1] Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza.
On verifying causal consistency.
In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 626–638. ACM, 2017.

References

[1] Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza.
On verifying causal consistency.
In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 626–638. ACM, 2017.