Typed Assembly Language for Implementing OS Kernels in SMP/Multi-Core Environments with Interrupts

Toshiyuki Maeda and Akinori Yonezawa

University of Tokyo
Quiz
Q. Can execution of the following 2 threads yield the result “\( r_1 = 0 \) and \( r_2 = 0 \)?

**Thread1:**
- `st [x] ← 1`
- `ld r1 ← [y]`

**Thread2:**
- `st [y] ← 1`
- `ld r2 ← [x]`

Initial state of shared memory:
- \( x: 0 \)
- \( y: 0 \)

[Environment]
- CPU: Intel Xeon X5570 (2.93GHz) x 8
- OS: Linux
Q. Can execution of the following 2 threads yield the result “r1 = 0 and r2 = 0”?

Thread1:
\[
\begin{align*}
\text{st} & \quad [x] \leftarrow 1 \\
\text{ld} & \quad r1 \leftarrow [y]
\end{align*}
\]

Thread2:
\[
\begin{align*}
\text{st} & \quad [y] \leftarrow 1 \\
\text{ld} & \quad r2 \leftarrow [x]
\end{align*}
\]

A. Yes
Quiz 2
Q. How often does it occur?

- 1. Once per second or more
- 2. Once a minute
- 3. Once an hour
- 4. Once a day
- 5. Once a month
- 6. Once a year (or less)

[Environment]
CPU: Intel Xeon X5570 (2.93GHz) x 8
OS: Linux
Q. How often does it occur?

A.

• 1. Once per second or more
• 2. Once a minute
• 3. Once an hour
• 4. Once a day
• 5. Once a month
• 6. Once a year (or less)

[Environment]
CPU: Intel Xeon X5570 (2.93GHz) x 8
OS: Linux
Typed Assembly Language for Implementing OS Kernels in SMP/Multi-Core Environments with Interrupts

or

Typed Assembly Language for Implementing Ad Hoc Synchronization Correctly
What is Typed Assembly Language (TAL)?

• “Strongly-typed” assembly language
  – Its type-checking ensures two safety
    • Memory safety
    • Control-flow safety

– Except for being typed, it is an ordinary assembly language

• It was first introduced in the field of type-preserving compilation [Morrisett et al. 1999]
Overview of TAL’s framework: generating binary executables

- TAL’s assembler generates not only binary executables, but also their type information.
Overview of TAL’s framework: type-checking binary executables

- TAL’s type-checker can type-check binary executables
  - utilizing type information generated by TAL’s assembler
TALK: TAL for Kernel [Maeda et al. 06, 08]

- TAL whose type system is extended in order to implement OS kernels
  - Memory management (malloc/free) and multi-thread management mechanisms can be written in TALK
    - It is impossible in conventional TALs
      - because they rely on external memory management (= GC)
Brief overview of TALK’s type system

• Supports variable-length arrays as a language primitive  
  – in order to represent memory regions

• Keeps track of integer constraints  
  (in the same way as dependent type [Xi et al. 99])  
  – in order to perform array-bound checking statically

• Keeps track of pointer aliases  
  (in the same way as alias type [Walker et al. 00])  
  – in order to realize safe strong update (explained in the next slide)

• Introduces notion of split/concatenation of arrays  
  – in order to integrate variable-length arrays and alias type
What is strong update?

• Memory operation that modifies types of memory regions
  – Memory management (e.g., malloc/free) can be viewed as strong updates
Memory management as strong update

• Example of memory reuse
  – Reusing “int*” as “int”

```c
int* reuse(int** o) {
    int* p = (int*)o;
    *p = 42;
    return p;
}
```
Memory management as strong update

• Example of memory reuse
  – Reusing “int*” as “int”

```c
int* reuse(int** o) {
  int* p = (int*)o;
  *p = 42;
  return p;
}
```

Reusing this region
Memory management as strong update

- Example of memory reuse
  - Reusing "int*" as "int"

```c
int* reuse(int** o) {
    int* p = (int*)o;
    *p = 42;
    return p;
}
```

Strong update
Memory management as strong update

• Example of memory reuse
  – Reusing “int*” as “int”

```c
int* reuse(int** o) {
    int* p = (int*)o;
    *p = 42;
    return p;
}
```

In general, strong updates are not always safe because pointer o may be used in other locations.
Memory management as strong update

- Example of memory reuse
  - Reusing “int*” as “int”

```c
int* reuse(int** o) {
    int* p = (int*)o;
    *p = 42;
    return p;
}
```

In general, strong updates are not always safe because pointer `o` may be used in other locations.

Alias type system ensures that this strong update is safe by ensuring that pointer `o` is not aliased with other pointers.
Problem of the original TALK

• The original alias type system becomes unsound in SMP/Multi-core environments
Why unsound?

• The original alias type system does not keep track of pointer aliases between threads

```c
int* reuse(int** o) {
    int* p = (int*)o;
    *p = 42;
    return p;
}
```

Unsafe

if pointer o is being used by other threads
An approach to making it sound

- Introduce synchronization primitives
  - Lock/unlock, synchronized block, atomic block, etc.

```c
int* reuse(int** o)
{
    lock(L);
    int* p = (int*)o;
    *p = 42;
    unlock(L);
    return p;
}
```
An approach to making it sound

• Introduce synchronization primitives
  – Lock/unlock, synchronized block, atomic block, etc.

```c
int* reuse(int* o) {
    lock(L);
    int* p = (int*)o;
    *p = 42;
    unlock(L);
    return p;
}
```

It doesn't work
Why doesn’t it work?

• Sync primitives don’t help for safe strong update
  – They can ensure race-freedom etc., but don’t tell whether types are changed or not

• Sync primitives are not available when implementing OS kernels
  – OS kernels should provide them by using low-level CPU instructions
Our approach to safe strong update in SMP/multi-core environments (1 of 2)

• Classify memory types into two kinds:
  – Local memory
    • Only a dedicated thread can access
  – Shared memory
    • Multiple threads can access
Our approach to safe strong update in SMP/multi-core environments (2 of 2)

- Local memory allows strong update
  - because other threads cannot access it

- Shared memory does not allow strong update
  - Except for a certain condition
When can we allow strong update of shared memory?

- If types of shared memory are invariant before and after execution of a CPU instruction
  - Strong updates are allowed between 1 CPU instruction + pseudo instructions
    - Pseudo instructions = Instructions that affect types only and have no runtime effects
When do we allow strong update of shared memory?

- If types of shared memory are invariant before and after execution of a CPU instruction
  - Strong updates are allowed between 1 CPU instruction + pseudo instructions
    - Pseudo instructions = Instructions that affect types only and have no runtime effects
When do we allow strong update of shared memory?

- If types of shared memory are invariant before and after execution of a CPU instruction
  - Strong updates are allowed during 1 CPU instruction + pseudo instructions
  - Pseudo instructions = Instructions that affect types only and have no runtime effects
When do we allow strong update of shared memory?

• If types of shared memory are invariant before and after execution of a CPU instruction
  – Strong updates are allowed during 1 CPU instruction + pseudo instructions
  – Pseudo instructions = Instructions that affect types only and have no runtime effects

With this approach, types appear to be invariant from the viewpoint of other threads
Example: lock acquisition

{p→ ∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)

lock:

    mov r2 ← 1
    unpack r1
    xchg [r1], r2
    pack r1
    bne r2, 0, lock

...
Example: lock acquisition

\{p \rightarrow \exists i. \{q \rightarrow \text{data if } [i==0]\}. (i, q)\}

(r1 : p)

lock:

- mov r2 \leftarrow 1
- unpack r1
- xchg [r1], r2
- pack r1
- bne r2, 0, lock

...
Example: lock acquisition

{p→ ∃i.{q→ data if [i==0]}. (i, q)}
(r1 : p)
lock:
   mov r2 ← 1
   unpack r1
   xchg [r1], r2
   pack r1
   bne r2, 0, lock
...

Thread

Address
p: (0, q)

Shared memory

Local memory

q: data
Example: lock acquisition

\{p \rightarrow \exists i.\{q \rightarrow \text{data if } [i==0]\}. (i, q)\}

(r1 : p)
lock:
  mov r2 \leftarrow 1
  unpack r1
  xchg [r1], r2
  pack r1
  bne r2, 0, lock
...

Address
p: (1, q)

Thread

q: data

Shared memory

Local memory
Example: type-checking lock acquisition

\{p \rightarrow \exists i. \{q \rightarrow \text{data if } [i==0]\}. \ (i, \ q)\} \\
(r1 : p) \\
lock: \\
\text{mov } r2 \leftarrow 1 \\
\text{unpack } r1 \\
\text{xchg } [r1], \ r2 \\
\text{pack } r1 \\
\text{bne } r2, \ 0, \ lock \\
\ldots

\text{State of the type checker} \\
\{p \rightarrow \exists i. \ldots (i, \ q)\} \\
(r1 : p, \ r2 : ??)
Example: type-checking lock acquisition

\[
\{ p \rightarrow \exists i. \{ q \rightarrow \text{data if } [i==0] \} \}. (i, q) \}
\]

(r1 : p)

lock:

\[
\begin{align*}
\text{mov} & \quad r2 \leftarrow 1 \\
\text{unpack} & \quad r1 \\
\text{xchg} & \quad [r1], r2 \\
\text{pack} & \quad r1 \\
\text{bne} & \quad r2, 0, \text{lock} \\
\end{align*}
\]

...
Example: type-checking lock acquisition

\{p \rightarrow \exists i. \{q \rightarrow \text{data if } [i == 0] \}. (i, q)\}

(r1 : p)

lock:

mov r2 \leftarrow 1
unpack r1
xchg [r1], r2
pack r1
bne r2, 0, lock
...

State of the type checker

\{p \rightarrow (i, q)\}
[q \rightarrow \text{data if } [i == 0]\]
(r1 : p, r2 : 1)
Example: type-checking lock acquisition

\{p \rightarrow \exists i. \{q \rightarrow \text{data if } [i == 0]\}. \ (i, q)\}

\[(r1 : p)\]

lock:

\[\text{mov } r2 \leftarrow 1\]
\[\text{unpack } r1\]
\[\text{xchg } [r1], \ r2\]
\[\text{pack } r1\]
\[\text{bne } r2, 0, \text{ lock}\]
...

State of the type checker:

\{p \rightarrow (i, q)\}
\[\text{q } \rightarrow \text{data}\]
\[\text{if } [i == 0]\]
\[(r1 : p, r2 : 1)\]

Strong update occurred:
The type has to be reverted before executing the CPU instruction after the next...
Example: type-checking lock acquisition

\[ \{ p \rightarrow \exists i. \{ q \rightarrow \text{data if } [i==0]\} \}. (i, q) \]  
\[ (r1 : p) \]

lock:

\[
\begin{align*}
\text{mov } r2 & \leftarrow 1 \\
\text{unpack } r1 \\
\text{xchg } [r1], r2 \\
\text{pack } r1 \\
\text{bne } r2, 0, \text{lock}
\end{align*}
\]

...
Example: type-checking lock acquisition

\{p \rightarrow \exists i.\{q \rightarrow \text{data if } [i==0]\}. (i, q)\}
(r1 : p)

lock:
- `mov r2 \leftarrow 1`
- `unpack r1`
- `xchg [r1], r2`
- `pack r1`
- `bne r2, 0, lock`

...
Example: type-checking lock acquisition

\[ \{ p \rightarrow \exists i. \{ q \rightarrow \text{data if } [i == 0] \}. (i, q) \} \]

(lock:
  mov r2 \leftarrow 1
  unpack r1
  xchg [r1], r2
  pack r1
  bne r2, 0, lock
  ...)

State of the type checker

\[ \{ p \rightarrow \exists i \ldots (i, q) \} \]

[\rightarrow \text{data if } [i == 0] \]

: p, r2 : i)

The type is revered correctly
Example: type-checking lock acquisition

\{p \rightarrow \exists i. \{q \rightarrow \text{data if } [i == 0]\}. \ (i, q)\}

\(r_1 : p\)

lock:

\[
\begin{align*}
\text{mov} & \quad r_2 \leftarrow 1 \\
\text{unpack} & \quad r_1 \\
\text{xchg} & \quad [r_1], r_2 \\
\text{pack} & \quad r_1 \\
\text{bne} & \quad r_2, 0, \text{lock}
\end{align*}
\]

...
Example: type-checking lock acquisition

\{p \rightarrow \exists i. \{q \rightarrow \text{data if } [i==0]\}. (i, q)\}

(r1 : p)

lock:

\begin{verbatim}
  mov r2 ← 1
  unpack r1
  xchg [r1], r2
  pack r1
  bne r2, 0, lock
  
  ...
\end{verbatim}

State of the type checker:

\{p \rightarrow \exists i \ldots. (i, q)\}

[q \rightarrow \text{data}]

(r1 : p, r2 : i)
Example: type-checking lock acquisition

\[
\{p \to \exists i.\{q \to \text{data if } [i=0]\}\}. (i, q)
\]

\(\text{lock:}\)

\[
\begin{align*}
\text{mov } r2 & \leftarrow 1 \\
\text{unpack } r1 \\
\text{xchg } [r1], r2 \\
\text{pack } r1 \\
\text{bne } r2, 0, \text{lock}
\end{align*}
\]

Succeed in extracting memory region \(q\) protected by a lock
Example: type-checking lock release

\[{p \rightarrow \exists i. \{q \rightarrow \text{data if } [i==0]\}. (i, q)}\]

[q \rightarrow \text{data}]

(r1 : p)

unlock:

  unpack r1

  mov [r1] \leftarrow 0

  pack r1

...
Example: type-checking lock release

\[ \{ p \rightarrow \exists i.\{ q \rightarrow \text{data if } [i==0]\}. (i, q) \} \]
\[ \text{[q \rightarrow \text{data}] (r1 : p)} \]

unlock:

- unpack r1
- mov [r1] \leftarrow 0
- pack r1
- ...

State of the type checker

\[ \{ p \rightarrow \exists i.\ldots.(i, q) \} \]
\[ \text{[q \rightarrow \text{data}] (r1 : p)} \]
Example: type-checking lock release

\{p \to \exists i. \{q \to \text{data if } [i==0]\}. (i, q)\} \quad \text{[q \to \text{data}]}
(r1 : p)
unlock:
  unpack r1
  mov [r1] \leftarrow 0
  pack r1
...

State of the type checker

\{p \to (i, q)\} \quad \text{[q \to \text{data}]}
\quad (r1 : p)
Example: type-checking lock release

\{p \rightarrow \exists i . \{q \rightarrow \text{data if } [i==0]\}\}. (i, q)\}

[q \rightarrow \text{data}]

(r_1 : p)

unlock:

\{\text{unpack } r_1\}
\{\text{mov } [r_1] \leftarrow 0\}
\{\text{pack } r_1\}
...

State of the type checker

\{p \rightarrow (i, q)\}

[q \rightarrow \text{data}]

[r_1 : p]

Strong update occurred:
The type has to be reverted before executing the CPU instruction after the next
Example: type-checking lock release

\{p \Rightarrow \exists i. \{q \Rightarrow \text{data if } [i==0]\}. (i, q)\}

[q \Rightarrow \text{data}]

(r1 : p)

unlock:

- unpack r1
- \text{mov } [r1] \leftarrow 0
- pack r1

...
Example: type-checking lock release

\{p \rightarrow \exists i. \{q \rightarrow \text{data if } [i=0] \}. (i, q)\} [q \rightarrow \text{data}]

(r1 : p)

unlock:
  unpack r1
  mov [r1] \leftarrow 0
  pack r1
...

State of the type checker

\{p \rightarrow \exists i. \ldots. (i, q)\}

(r1 : p)
Example: type-checking lock release

\{p \rightarrow \exists i. \{q \rightarrow \text{data if } [i==0]\}. (i, q)\} \\
[q \rightarrow \text{data}] \\
(r1 : p) \\
\text{unlock:} \\
\text{unpack r1} \\
\text{mov [r1] } \leftarrow 0 \\
\text{pack r1} \\
... \\

The type is revered correctly and memory region q is successfully returned back to shared memory.
About CPU interrupts

• CPU interrupts can be type-checked in a similar way
  • Interrupt handlers and interrupted programs can be viewed as concurrently running threads
  – Strong update is basically not allowed to shared memory between interrupters/interruptees
  • If interrupts are disabled using CPU’s interrupt flag, strong updates are allowed on the shared memory
One limitation of our approach explained so far

- Relaxed memory models of today’s CPU are not considered
  - Shared memory of relaxed memory consistency may violate memory safety property
What is relaxed memory consistency?

• In short, memory consistency models that allow effects of memory operations on one CPU to be observed in a different order from other CPUs.
Example of relaxed memory consistency

• Execution of the following 2 threads can yield the result “\( r_1 = 0 \) and \( r_2 = 0 \)"

**Thread1:**

\[
\begin{align*}
st \ [x] & \leftarrow 1 \\
ld \ r1 & \leftarrow [y]
\end{align*}
\]

**Thread2:**

\[
\begin{align*}
st \ [y] & \leftarrow 1 \\
ld \ r2 & \leftarrow [x]
\end{align*}
\]

Initial state of shared memory:

- Address:
  - \( x: 0 \)
  - \( y: 0 \)
Example of relaxed memory consistency

- Execution of the following 2 threads can yield the result "\( r_1 = 0 \) and \( r_2 = 0 \)"

Thread1:
- \( \text{st} \ [x] \leftarrow 1 \)
- \( \text{ld} \ r1 \leftarrow [y] \)

Thread2:
- \( \text{st} \ [y] \leftarrow 1 \)
- \( \text{ld} \ r2 \leftarrow [x] \)

Effects of these instructions may be reordered in Thread1

Effects of these instructions may be reordered in Thread2
Example of relaxed memory consistency

• Execution of the following 2 threads can yield the result “\( r_1 = 0 \) and \( r_2 = 0 \)”

```
Thread1:
  st [x] \leftarrow 1
  ld r1 \leftarrow [y]

Thread2:
  st [y] \leftarrow 1
  ld r2 \leftarrow [x]
```

Effects of these instructions may be reordered in Thread2
Example of relaxed memory consistency

• Execution of the following 2 threads can yield the result “r1 = 0 and r2 = 0”

Thread1:

```
st [x] ← 1
ld r1 ← [y]
```

Address

y: 0
x: 0

Thread2:

```
st [y] ← 1
ld r2 ← [x]
```

Effects of these instructions may be reordered in Thread1
How to control memory reordering in relaxed memory consistency models?

• Typically, utilize two mechanisms provided by today’s CPUs

  – Atomic memory operation mechanism
    • E.g., “lock” prefix on Intel Architecture

  – Memory ordering control mechanism
    • E.g., acquire/release
Atomic memory operation

• Memory operation whose effect is observed in an “all-or-nothing” way by other threads
Memory ordering control

- Acquire operation
  - Operation whose effect becomes observable from other threads before any succeeding operation

- Release operation
  - Operation whose effect becomes observable from other threads after any preceding operation
Example of memory ordering control

- Execution of the following 2 threads never yields the result “r1 = 0 and r2 = 0”

Thread1:
- st [x] ← 1
- release
- acquire
- ld r1 ← [y]

Thread2:
- st [y] ← 1
- release
- acquire
- ld r2 ← [x]

Initial state of shared memory:
- x: 0
- y: 0
Example of memory ordering control

- Execution of the following 2 threads never yields the result “\( r_1 = 0 \) and \( r_2 = 0 \)”
Our type-checking approach in order to support relaxed memory consistency (just an idea)

• Check the following 2 constraints with type system
  
  – Only atomic memory operations are able to perform strong update on shared memory
  
  – Memory ordering control mechanisms are used properly when moving memory regions between shared memory and local memory
    – Shared memory → local memory: use acquire
    – Local memory → shared memory: use release
Example of lock acquisition in a relaxed memory consistency model

\{p \rightarrow \exists i \cdot \{q \rightarrow \text{data if } [i==0]\} \cdot (i, q)\}

(r1 : p)
lock:

mov r2 \leftarrow 1
unpack r1
atomic xchg [r1], r2
acquire
pack r1
bne r2, 0, lock
...

Example of type-checking lock acquisition in a relaxed memory consistency model

\{p \rightarrow \exists i. \{q \rightarrow \text{data if } [i==0]\}. \ (i, \ q)\}

(r1 : p)

lock:

- mov r2 \leftarrow 1
- unpack r1
- atomic xchg [r1], r2
- acquire
- pack r1
- bne r2, 0, lock
- ...

State of the type-checker

\{p \rightarrow \exists i.\ldots.\ (i, \ q)\}

(r1 : p, r2 : ??)
Example of type-checking lock acquisition in a relaxed memory consistency model

\[
\{p \rightarrow \exists i.\{q \rightarrow \text{data if } [i==0]\}\}. \ (i, \ q)\}
\]

\(r1 : p\)

lock:

\begin{align*}
\text{mov } r2 & \leftarrow 1 \\
\text{unpack } r1 \\
\text{atomic } \text{xchg } [r1], r2 \\
\text{acquire} \\
\text{pack } r1 \\
\text{bne } r2, \ 0, \ \text{lock} \\
\end{align*}

\(\ldots\)

State of the type-checker

\[
\{p \rightarrow \exists i.\ldots.(i, \ q)\}
\]

\(r1 : p, \ r2 : 1\)
Example of type-checking lock acquisition in a relaxed memory consistency model

\{ p \rightarrow \exists i. \{ q \rightarrow \text{data if } [i==0] \}. (i, q) \} (r1 : p)

lock:
  mov r2 \leftarrow 1
  unpack r1
  atomic xchg [r1], r2
  acquire
  pack r1
  bne r2, 0, lock
...

State of the type-checker

\{ p \rightarrow (i, q) \}
[q \rightarrow \text{data if } [i == 0]] (r1 : p, r2 : 1)
Example of type-checking lock acquisition in a relaxed memory consistency model

\{p \rightarrow \exists i. \{q \rightarrow \text{data if } [i == 0]\}. (i, q)\}

(r1 : p)
lock:

- mov r2 ← 1
- unpack r1
- atomic xchg [r1], r2
- acquire
- pack r1
- bne r2, 0, lock
...

State of the type-checker

\{p \rightarrow (i, q)\}

[q \rightarrow \text{data if } [i == 0]\}

1 : p, r2 : 1)

Memory region q is still not accessible because acquire is not performed
Example of type-checking lock acquisition in a relaxed memory consistency model

\{p \rightarrow \exists i. \{q \rightarrow \text{data if } [i == 0]\}. \ (i, \ q)\}

(r1 : p)

lock:

    mov r2 \leftarrow 1
    unpack r1
    atomic \text{xchg} [r1], r2
    acquire
    pack r1
    bne r2, 0, lock
...
Example of type-checking lock acquisition in a relaxed memory consistency model

\{p \rightarrow \exists i. \{q \rightarrow \text{data if } [i == 0]\}. (i, q)\}
(r1 : p)
lock:

- `mov r2 \leftarrow 1`
- `unpack r1`
- `atomic xchg [r1], r2`
- `acquire`
- `lock r1`
- `the r2, 0, lock`

This memory operation on a shared memory region is OK because it is atomic

State of the type-checker:

\{p \rightarrow (1, q)\}
[q \rightarrow \text{data if } [i == 0]]
(r1 : p, r2 : i)
Example of type-checking lock acquisition in a relaxed memory consistency model

\{p \rightarrow \exists i. \{q \rightarrow \text{data if } [i == 0]\}. (i, q)\}

(r1 : p)

\textbf{lock:}

\begin{align*}
\text{mov } r2 & \leftarrow 1 \\
\text{unpack } r1 \\
\text{atomic } xchg [r1], r2 \\
\text{acquire} \\
\text{pack } r1 \\
\text{bne } r2, 0, \text{lock}
\end{align*}

\ldots

\text{State of the type-checker}

\{p \rightarrow (1, q)\}
[q \rightarrow \text{data if } [i == 0]]
(r1 : p, r2 : i)
Example of type-checking lock acquisition in a relaxed memory consistency model

$$\{ p \rightarrow \exists i. \{ q \rightarrow \text{data} \text{ if } [i==0] \} \}. \ (i, q) \}$$

(r1 : p)

lock:

    mov r2 \leftarrow 1
    unpack r1
    atomic xchg [r1], r2
    acquire
    pack r1
    bne r2, 0, lock

...
Example of type-checking lock acquisition in a relaxed memory consistency model

\[\{p \rightarrow \exists i. \{q \rightarrow \text{data if } [i == 0]\}\}. (i, q)\]

(lock : p)

\[
\text{mov } r2 \leftarrow 1 \\
\text{unpack } r1 \\
\text{atomic } \text{xchg } [r1], r2 \\
\text{acquire} \\
\text{pack } r1 \\
\text{bne } r2, 0, \text{lock} \\
... \\
\]

State of the type-checker:

\[
\{p \rightarrow \exists i.....(i, q)\} \\
[q \rightarrow \text{data} \\
\text{if } [i == 0]] \\
(r1 : p, r2 : i) \\
\]
Example of type-checking lock acquisition in a relaxed memory consistency model

\[ \{ p \rightarrow \exists i. \{ q \rightarrow \text{data if } [i==0] \} \}. \ (i, q) \]  
\( (r1 : p) \)

lock:

- mov r2 ← 1
- unpack r1
- atomic xchg [r1], r2
- acquire
- pack r1
- bne r2, 0, lock
- ...

State of the type-checker:

\[ \{ p \rightarrow \exists i. \ldots \}. \ (i, q) \]  
\[ [q \rightarrow \text{data if } [i == 0]] \]  
\( (r1 : p, r2 : i) \)
Example of type-checking lock acquisition in a relaxed memory consistency model

\[
\{ p \to \exists i. \{ q \to \text{data if } [i==0] \} \}. (i, q) \\
(r_1 : p) \\
\text{lock:} \\
\begin{align*}
\text{mov } r_2 & \leftarrow 1 \\
\text{unpack } r_1 \\
\text{atomic } xchg \ [r_1], r_2 \\
\text{acquire} \\
\text{pack } r_1 \\
\text{bne } r_2, 0, \text{ lock} \\
\end{align*}
\]

State of the type-checker

\[
\{ p \to \exists i .... (i, q) \} \\
[q \to \text{data}] \\
(r_1 : p, r_2 : i)
\]
Example of type-checking lock acquisition in a relaxed memory consistency model

\[ \{ p \to \exists i. \{ q \to \text{data if } [i==0]\} \}. (i, q) \] 
(r1 : p)

lock:
  mov r2 ← 1
  unpack r1
  atomic xchg [r1], r2
  acquire
  pack r1
  bne r2, 0, lock
...

State of the type-checker

\[ \{ p \to \exists i.....(i, q) \} \] 
[ q \to \text{data} ]
(r1 : p, r2 : i)

Succeed in extracting memory region q protected by a lock
Related work (1/3)

• Type-based approaches
  – A multithreaded typed assembly language
    [Vasconcelos et al. 2006]
    • It cannot be used to implement synchronization primitives and multi-thread management mechanisms themselves
      – Mutex locks and threading mechanisms are provided as language primitives
  – Type-based analysis of synchronization lock usage
    • They cannot be used to analyze synchronization primitives and multi-thread management mechanisms themselves
      – Their goals are to ensure race/deadlock- freedom
        » whereas our goal is limited to ensuring simple type safety
Related work (2/3)

• Separation logic approaches
  – **Abstract Interrupt Machine (AIM)**
    [Feng et al. 2008]
    • Utilizing separation logic in order to verify programs with CPU interrupts by maintaining invariants on interrupters/interruptees
    – SMP/multi-core environments are not considered
  – **Concurrent Abstract Predicates**
    [Dinsdale-Young et al. 2010]
    • Utilizing separation logic in order to handle invariants on shared memory between multiple threads
    – Relaxed memory consistency models are not considered
Related work (3/3)

• Program verification for relaxed memory consistency models
  – **Sober** [Burckhardt et al. 2008]
    • A bounded model checker that checks whether a program on TSO satisfies SC
    • Define semantics of relaxed memory models in operational-semantics styles for program verification
Conclusion and future work

• We presented Typed Assembly Language for SMP/multi-core environments with CPU interrupts
  – Memory and control-flow safety can be verified
  – Sync primitives can be directly written in it
    • We also showed an idea of how to support relaxed memory consistency models

• Future work:
  – Prove the soundness of our type system, particularly for the extension of relaxed memory models
  – Implement an OS kernel with our TAL
  – Extend the type system further in order to support more complex and efficient synchronization primitives