# Static Versioning of Global State for Race Condition Detection

Steffen Keul

Dept. of Programming Languages and Compilers
Institute of Software Technology
University of Stuttgart

15th International Conference on Reliable Software Technologies – Ada-Europe 2010

University of Stuttgart
Germany

# Outline

# Example: Real-World Data Race

```
int control()
{
  ...
  if (sensor_valid &&
      sensor >= min_threshold &&
      sensor <= max_threshold)
  {
    control_car(sensor);
    ...
  }
}
```

# Example: Real-World Data Race

```
int sensor;

int control()
{
  ...
  if (sensor_valid &&
      sensor >= min_threshold &&
      sensor <= max_threshold)
  {
    control_car(sensor);
    ...
  }
}
```

```
interrupt void isr()
{
  ...
  sensor_valid =
    read_port(0x23, &sensor);
}
```

# Example: Real-World Data Race

```
int sensor;

int control()
{
  ...
  if (sensor_valid &&
      sensor >= min_threshold &&
      sensor <= max_threshold)
  {
    control_car(sensor);
    ...
  }
}
```

```
interrupt void isr()
{
  ...
  sensor_valid =
    read_port(0x23, &sensor);
}
```

# Example: Real-World Data Race

```
int sensor;

int control()
{
  ...
  if (sensor_valid &&
     sensor >= min_threshold &&
     sensor <= max_threshold)
  {
    control_car(sensor);
    ...
  }
}
```

consistent
data?

```
interrupt void isr()
{
  ...
  sensor_valid =
    read_port(0x23, &sensor);
}
```

# Data Races

## Definition (Data Race)

A data race occurs if two threads access a common storage location without ordering constraints, and one of the accesses modifies the storage contents.

Presence of data race means:

- possibly missing explicit synchronization
- for non-atomic accesses, possibility of illegal bit-patterns

Absence of data race means:

- some serialization of accesses exists
- no illegal bit-patterns are created

# Race detection

- data races can indicate programming errors
- confidence in absence of races through static analysis
- many analysis algorithms exist for data race detection
- some data races can be tolerated if the shared variable is accessed atomically
- however, some critical race conditions are not data races
- this work aims at detection of all potentially harmful race conditions

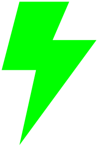
# Example: Static State Versioning

```
int f()
{
  lock(&m);
  int l1 = sens_1;
  int l2 = sens_2;
  unlock(&m);
  ...
  int l3 = sens_3;
  ...
  if (l1 < l2) ...;
  ...
  if (l2 < l3) ...;
}
```

- Shared Variables: `sens_1`, `sens_2`, `sens_3`

# Example: Static State Versioning

```
int f()
{
  lock(&m);
  int l1 = sens_1;
  int l2 = sens_2;
  unlock(&m);
  ...
  int l3 = sens_3;
  ...
  if (l1 < l2) ...;
  ...
  if (l2 < l3) ...;
}
```



Data Race!
Problem?

- Shared Variables: `sens_1`, `sens_2`, `sens_3`
- Data Race because of read of `sens_3`
- no synchronization necessary if **`int`**s read atomically, Data Race uninteresting

# Example: Static State Versioning

```
int f()
{
  lock(&m);              version 1
  int l1 = sens_1;
  int l2 = sens_2;
  unlock(&m);
  ...
  int l3 = sens_3;       version 2
  ...
  if (l1 < l2) ...;
  ...
  if (l2 < l3) ...;
}
```

- Shared Variables: `sens_1`, `sens_2`, `sens_3`
- Data Race because of read of `sens_3`
- no synchronization necessary if **`int`**s read atomically, Data Race uninteresting
- Versioning of reads

# Example: Static State Versioning

```
int f()
{
  lock(&m);            version 1
  int l1 = sens_1;
  int l2 = sens_2;
  unlock(&m);
  ...
  int l3 = sens_3;     version 2
  ...
  if (l1 < l2) ...;
  ...
  if (l2 < l3) ...;
}
```

consistent?

- Shared Variables: `sens_1`, `sens_2`, `sens_3`
- Data Race because of read of `sens_3`
- no synchronization necessary if **`ints`** read atomically, Data Race uninteresting
- Versioning of reads
- Use of different versions indicates programming error

# Violation of Atomicity: uninteresting warnings

**Example (Conflict accesses on `g` in `thread2` and `thread3`, but inconsistent expression only in `thread3`)**

```
int g;

void *thread1(void *p)
{   while (1) g = read_sensor_value();   }

void *thread2(void *p)
{   while (1) act_1(5 * g + 17);   }

void *thread3(void *p)
{   while (1) act_2(g * g);   }
```

# Violation of Atomicity: nonatomic expressions

**Example (Free of data races, but the `mutex_lock`-calls around `g1+g2` have no effect)**

```
void *t1(void *p)         void *t2(void *p)
{ mutex_lock(&m);         { mutex_lock(&n);
  g1 = ...;                 g2 = ...;
  mutex_unlock(&m);         mutex_unlock(&n);
}                         }

int main()
{  create(t1); create(t2);
   mutex_lock(&m); mutex_lock(&n);
   res = g1 + g2;
   mutex_unlock(&n); mutex_unlock(&m);
}
```

# Stale Updates

## Example (Nonatomic increments)

```
pthread_mutex_lock(&m);
int local = global;
pthread_mutex_unlock(&m);

local += 17;

pthread_mutex_lock(&m);
global = local;
pthread_mutex_unlock(&m);
```

The LHS's version (`global` directly before the assignment) differs from the RHS's version (`local`).

# State Versioning Algorithm

1. translate source code into intermediate representation, use only atomic read and write operations
2. represent interfering data flow explicitly by insertion of $\psi$-nodes for
   - conflict reads
   - uses of shared variables in protected regions
3. assign versions to reads in every function independent of calling context, in bottom-up traversal of the call graph
4. adjust versions depending on context in top-down traversal of the call graph
5. produce warning list for potentially inconsistent expressions

# Lockset analysis

- determine the set of all possible (mutex-) locks: $L_{\text{full}}$
- associate each site $s$ in the program with the set of mutex-locks $l_{\text{act}}(s) \subseteq L_{\text{full}}$ that are active
  - use monotonic analysis framework over $(2^{L_{\text{full}}}, \subseteq)$
  - initial value $\varnothing$ at function entry, $L_{\text{full}}$ for all other basic blocks
  - at confluence points use intersection as meet operator
  - distinguish different caller locksets at call sites

# Interference flow for conflict reads

- determine shared objects
- use locksets to determine conflict reads
- place $\psi$-node in front of every conflict read

## Example (Insertion of $\psi$-nodes for conflict reads)

$$
\begin{array}{lcl}
\texttt{s = 0;} & & s_{m1} = 0; \\
 & \Rightarrow & s_{m2} = \psi(s_{m1}, \boxed{s_{t1}}, \ldots, \boxed{s_{tn}}); \\
 & & s_{m3} = \psi(s_{m1}, \boxed{s_{t1}}, \ldots, \boxed{s_{tn}}); \\
\texttt{s = s + s;} & & s_{m4} = s_{m2} + s_{m3};
\end{array}
$$

So far ...

- Synchronization is ignored

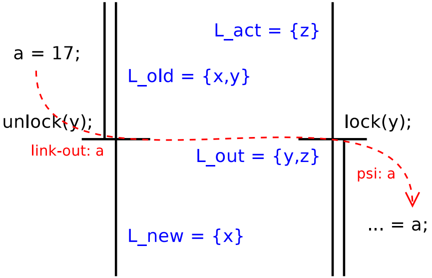
# Interference flow for protected regions

- identify protected regions
- regions protected by a common lock are mutually exclusive
- data flow can only occur from end to beginning of mutually exclusive regions

$\Rightarrow$ Add Link-out and $\psi$ nodes

- interference flow for multiple objects is stored into a single $\psi$-node

$$
\begin{aligned}
A = \{(l_{\text{old}}, l_{\text{new}}) \in 2^{L_{\text{full}}} \times 2^{L_{\text{full}}} :\ & l_{\text{old}} \cap \mathrm{L}_{\text{out}}(bb) \neq \varnothing \\
\wedge \quad & l_{\text{new}} \cap \mathrm{L}_{\text{out}}(bb) = \varnothing \\
\wedge \quad & l_{\text{old}} \cap \mathrm{L}_{\text{act}}(bb) = \varnothing \\
\wedge \quad & l_{\text{new}} \cap \mathrm{L}_{\text{act}}(bb) = \varnothing \}
\end{aligned}
$$



# State Version Analysis

- every execution of a $\psi$-node represents a unique observation of global state
- a unique version is assigned to every observation
- versions are propagated along the data flow paths
- every expression is assigned a version based on the versions that flow into the expression
- if values of more than one version flow into an expression, it is considered potentially inconsistent

# Bottom-up Pass

- state space: set of mappings **Var** $\rightarrow \{\perp, \top, \psi_1, \ldots, \psi_n\}$
- optimistic assumption: caller does not propagate versions into callee function
- analyze functions separately in reverse topological order
- multiple iterations for loops and recursion until fixed point is reached
- transfer function propagates versions across copy-statements
- if a node $a = \psi_i$ is encountered, all variables of version $i$ are set to $\perp$ and $a$'s version is set to $i$
- at call sites, use result of callee's analysis, treat every version $j$ of the callee like an encounter of a node $\psi_j$

# Top-down Pass

- use the active state at a call site to propagate versions into the callee function
- propagate versions along the def-use data flow links inside the callee to update versions
- contexts at different call sites can be distinguished or can be joined before the propagation

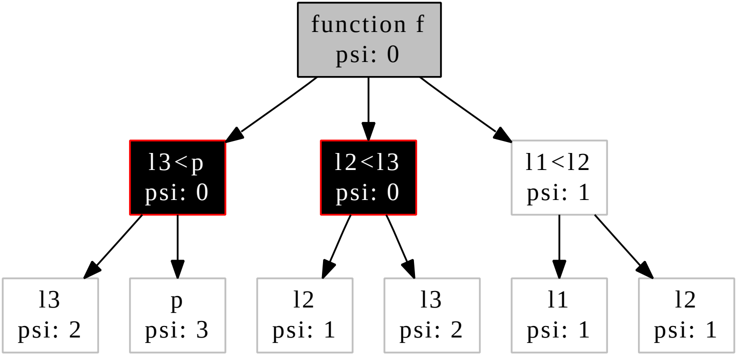
| | l1 | l2 | l3 | p | xpr |
|---|---|---|---|---|---|
| `void f(int p)` | $\top$ | $\top$ | $\top$ | $\top$ | |
| `{` | | | | | |
| `lock(&m);` | | | | | |
| `sens_1,2 =` $\psi_1$`(sens_1,2,...);` | | | | | |
| `int l1 = sens_1;` | 1 | $\top$ | $\top$ | $\top$ | |
| `int l2 = sens_2;` | 1 | 1 | $\top$ | $\top$ | |
| `unlock(&m);` | | | | | |
| `...` | | | | | |
| `sens_3 =` $\psi_2$`(sens_3,...);` | | | | | |
| `int l3 = sens_3;` | 1 | 1 | 2 | $\top$ | |
| `if (l1 < l2) ...;` | 1 | 1 | 2 | $\top$ | |
| `if (l2 < l3) ...;` | 1 | 1 | 2 | $\top$ | |
| `if (l3 < p) ...;` | 1 | 1 | 2 | $\top$ | |
| `}` | | | | | |

| | `l1` | `l2` | `l3` | `p` | xpr |
|---|---|---|---|---|---|
| **`void`** `f(`**`int`** `p)` | $\top$ | $\top$ | $\top$ | $\top$ | |
| `{` | | | | | |
| `lock(&m);` | | | | | |
| `sens_1,2 =` $\psi_1$`(sens_1,2,...);` | | | | | |
| **`int`** `l1 = sens_1;` | 1 | $\top$ | $\top$ | $\top$ | |
| **`int`** `l2 = sens_2;` | 1 | 1 | $\top$ | $\top$ | |
| `unlock(&m);` | | | | | |
| `...` | | | | | |
| `sens_3 =` $\psi_2$`(sens_3,...);` | | | | | |
| **`int`** `l3 = sens_3;` | 1 | 1 | 2 | $\top$ | |
| **`if`** `(l1 < l2) ...;` | 1 | 1 | 2 | $\top$ | 1 |
| **`if`** `(l2 < l3) ...;` | 1 | 1 | 2 | $\top$ | $\perp$ |
| **`if`** `(l3 < p) ...;` | 1 | 1 | 2 | $\top$ | 2 |
| `}` | | | | | |

| f($\psi_3$(...)); | l1 | l2 | l3 | p | xpr |
|---|---|---|---|---|---|
| **void** f(**int** p) | $\top$ | $\top$ | $\top$ | 3 | |
| { | | | | | |
| lock(&m); | | | | | |
| sens_1,2 = $\psi_1$(sens_1,2,...); | | | | | |
| **int** l1 = sens_1; | 1 | $\top$ | $\top$ | $\top$ | |
| **int** l2 = sens_2; | 1 | 1 | $\top$ | $\top$ | |
| unlock(&m); | | | | | |
| ... | | | | | |
| sens_3 = $\psi_2$(sens_3,...); | | | | | |
| **int** l3 = sens_3; | 1 | 1 | 2 | $\top$ | |
| **if** (l1 < l2) ...; | 1 | 1 | 2 | $\top$ | 1 |
| **if** (l2 < l3) ...; | 1 | 1 | 2 | $\top$ | $\perp$ |
| **if** (l3 < p) ...; | 1 | 1 | 2 | 3 | $\perp$ |
| } | | | | | |

# State Versioning Output

- warnings on possibly inconsistent expressions
- displayed in their syntactical context
- warnings on same combination of versions are output only once



# Evaluation

- implementation of the analysis in the Bauhaus system
- able to handle larger programs
  - clamd: 66 KSLoC
  - full context sensitivity needs 15 min
  - 6,667 warnings
- number of warnings
  - precision in data flow relation important
  - flow-insensitive points-to information
  - recognition of reference parameters not yet implemented
- future work
  - increase precision in data flow representation
  - determine cut-off strategy for data flow chains

# Conclusion

- new analysis algorithm to detect inconsistent uses
- can find error patterns that data race detectors cannot
- can deal with atomic accesses
- generates higher quality warnings, easier to validate
- future work to deal with precision