Implementation of a Type and Effect System for Deadlock Avoidance in C/Pthreads

Panagiotis Vekris

School of Electrical and Computer Engineering
National Technical University of Athens

We are going to talk about...

- Concurrency – deadlocks – deadlock freedom
- Our apprach to deadlock avoidance
- Deadlock avoidance tool
- Performance evaluation

Concurrent programming

- Computational processes executed in parallel

Concurrent programming

- Computational processes executed in parallel

Implementation

- Processes:

  independent execution units with their own state and address space

- Threads:

  multiple threads within a process, sharing memory

Concurrent programming

- Computational processes executed in parallel

Implementation

- Processes:

  independent execution units with their own state and address space

- Threads:

  multiple threads within a process, sharing memory

Interaction/Communication

- Shared Memory:
  - common address space, needs synchronization
  - e.g. Pthreads, Java
- Message Passing:
  - cooperating processes: send, receive messages
  - e.g. MPI, Erlang

Concurrent programming

- Computational processes executed in parallel

Implementation

- Processes:

  independent execution units with their own state and address space

- Threads:

  multiple threads within a process, sharing memory

Interaction/Communication

- Shared Memory:
  - common address space, needs synchronization
  - e.g. Pthreads, Java
- Message Passing:
  - cooperating processes: send, receive messages
  - e.g. MPI, Erlang

Concurrency hazards

Synchronization related

Concurrency hazards

Synchronization related

Data Races

- Failure to protect shared data

Concurrency hazards

Synchronization related

Data Races

- Failure to protect shared data

Deadlocks

- Mutual exclusion
- Hold and wait
- No preemtion
- Circular wait

  two or more threads form a circular chain, where each thread waits for a lock held by the next thread in chain

Concurrency hazards

Synchronization related

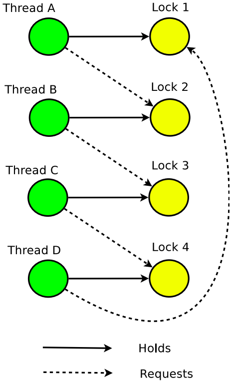
Data Races

- Failure to protect shared data

Deadlocks

- Mutual exclusion
- Hold and wait
- No preemtion
- Circular wait

  two or more threads form a circular chain, where each thread waits for a lock held by the next thread in chain



Deadlock Freedom

Prevention

Ensure that at least one deadlock condition is not satisfied.

Deadlock Freedom

Prevention

Ensure that at least one deadlock condition is not satisfied.

Detection and Recovery

Preempt some of the deadlocked threads (e.g. release some of their locks)

Deadlock Freedom

Prevention

Ensure that at least one deadlock condition is not satisfied.

Detection and Recovery

Preempt some of the deadlocked threads (e.g. release some of their locks)

Avoidance

- Use statically computed information regarding thread resource allocation
- Determine whether granting a lock will bring the program to an unsafe state.

Deadlock Freedom

Prevention

Ensure that at least one deadlock condition is not satisfied.

Detection and Recovery

Preempt some of the deadlocked threads (e.g. release some of their locks)

Avoidance

- Use statically computed information regarding thread resource allocation
- Determine whether granting a lock will bring the program to an unsafe state.

Deadlock Prevention: too restrictive

Key idea:

- impose a single global lock order
- check that all threads respect this lock order

Deadlock Prevention: too restrictive

Key idea:

- impose a single global lock order
- check that all threads respect this lock order

Most type-based approaches fall into this strategy

- a type and effect system is used
- effects record the lock acquisition order

Deadlock Prevention: too restrictive

Key idea:

- impose a single global lock order
- check that all threads respect this lock order

Most type-based approaches fall into this strategy

- a type and effect system is used
- effects record the lock acquisition order

$(\texttt{lock}\ x\ ;\ \ldots\ \texttt{lock}\ y\ ;\ \ldots)\ ||\ (\texttt{lock}\ y\ ;\ \ldots\ \texttt{lock}\ x\ ;\ \ldots)$

Deadlock Prevention: too restrictive

Key idea:

- impose a single global lock order
- check that all threads respect this lock order

Most type-based approaches fall into this strategy

- a type and effect system is used
- effects record the lock acquisition order

$$\underbrace{(\texttt{lock}\, x\, ;\ \ldots\ \texttt{lock}\, y\, ;\ \ldots)}_{x \leq y} \;||\; \underbrace{(\texttt{lock}\, y\, ;\ \ldots\ \texttt{lock}\, x\, ;\ \ldots)}_{y \leq x}$$

- no single global order $\Rightarrow$ reject program

Deadlock Avoidance: Boudol's Approach

Admits more programs by lifting the "ordering" restriction

Deadlock Avoidance: Boudol's Approach

Admits more programs by lifting the “ordering” restriction

Key idea:

- statically: for each lock operation compute the “future lockset”
- dynamically: check that the “future lockset” is available before granting the lock

Future lockset: the set of locks that will be obtained before this lock is released

Boudol's Approach: Example

$$(\texttt{lock}\,x\,;\quad \ldots\ \texttt{lock}\,y\,;\ \ \ldots) \parallel (\texttt{lock}\,y\,;\quad \ldots\ \texttt{lock}\,x\,;\ \ \ldots)$$

Boudol's Approach: Example

$$(\texttt{lock}_{\{y\}}\, x\, ;\, \underbrace{\ldots\ \texttt{lock}_{\emptyset}\, y\, ;\ \ldots}_{\text{only } y \text{ is locked here}})\ \|\ (\texttt{lock}_{\{x\}}\, y\, ;\, \underbrace{\ldots\ \texttt{lock}_{\emptyset}\, x\, ;\ \ldots}_{\text{only } x \text{ is locked here}})$$

Boudol's Approach: Example

$$(\,\texttt{lock}_{\{y\}}\ x\ ;\ \underbrace{\dots\ \texttt{lock}_{\emptyset}\ y\ ;\ \dots}_{\text{only } y \text{ is locked here}}\,)\ \ ||\ \ (\,\texttt{lock}_{\{x\}}\ y\ ;\ \underbrace{\dots\ \texttt{lock}_{\emptyset}\ x\ ;\ \dots}_{\text{only } x \text{ is locked here}}\,)$$

At runtime, the lock annotation is checked

Boudol's Approach: Example

$$(\texttt{lock}_{\{y\}}\ x\ ;\ \underbrace{\ldots\ \texttt{lock}_{\emptyset}\ y\ ;\ \ldots}_{\text{only } y \text{ is locked here}})\ \|\ (\texttt{lock}_{\{x\}}\ y\ ;\ \underbrace{\ldots\ \texttt{lock}_{\emptyset}\ x\ ;\ \ldots}_{\text{only } x \text{ is locked here}})$$

At runtime, the lock annotation is checked

- thread 1 tries to lock x, with future lockset $\{y\}$

Boudol's Approach: Example

$$(\,\texttt{lock}_{\{y\}}\ x\ ;\ \underbrace{\ldots\ \texttt{lock}_{\emptyset}\ y\ ;\ \ldots}_{\text{only } y \text{ is locked here}}\,)\ \|\ (\,\texttt{lock}_{\{x\}}\ y\ ;\ \underbrace{\ldots\ \texttt{lock}_{\emptyset}\ x\ ;\ \ldots}_{\text{only } x \text{ is locked here}}\,)$$

At runtime, the lock annotation is checked

- thread 1 tries to lock x, with future lockset $\{y\}$ success!

Boudol's Approach: Example

$$(\,\texttt{lock}_{\{y\}}\ \boxed{x}\ ;\ \underbrace{\ldots\ \texttt{lock}_{\emptyset}\ y\ ;\ \ldots}_{\text{only } y \text{ is locked here}}\,)\ \|\ (\,\boxed{\texttt{lock}_{\{x\}}\ y\ ;}\ \underbrace{\ldots\ \texttt{lock}_{\emptyset}\ x\ ;\ \ldots}_{\text{only } x \text{ is locked here}}\,)$$

At runtime, the lock annotation is checked

- thread 1 tries to lock x, with future lockset $\{y\}$ success!
- thread 2 tries to lock y, with future lockset $\{x\}$

Boudol's Approach: Example

$$(\texttt{lock}_{\{y\}}\ \boxed{x}\ ;\ \underbrace{\ldots\ \texttt{lock}_{\emptyset}\ y\ ;\ \ldots}_{\text{only } y \text{ is locked here}}) \parallel (\boxed{\texttt{lock}_{\{x\}}\ \boxed{y}\ ;}\ \underbrace{\ldots\ \texttt{lock}_{\emptyset}\ x\ ;\ \ldots}_{\text{only } x \text{ is locked here}})$$

At runtime, the lock annotation is checked

- thread 1 tries to lock x, with future lockset $\{y\}$ — success!
- thread 2 tries to lock y, with future lockset $\{x\}$ — block!

Boudol's Approach: Example

$$(\,\texttt{lock}_{\{y\}}\ \boxed{x}\,;\ \underbrace{\ldots\ \texttt{lock}_{\emptyset}\ y\ ;\ \ldots}_{\text{only } y \text{ is locked here}}\,)\ \|\ (\,\texttt{lock}_{\{x\}}\ \boxed{y}\,;\ \underbrace{\ldots\ \texttt{lock}_{\emptyset}\ x\ ;\ \ldots}_{\text{only } x \text{ is locked here}}\,)$$

At runtime, the lock annotation is checked

- thread 1 tries to lock x, with future lockset $\{y\}$ success!
- thread 2 tries to lock y, with future lockset $\{x\}$ block!

Lock y is available, but lock x is held by thread 1

- granting y to thread 2 may lead to a deadlock !

Boudol's Approach: Drawbacks

But, Boudol's approach is context insensitive and does not support unstructured locking.

Boudol's Approach: Drawbacks

But, Boudol's approach is context insensitive and does not support unstructured locking.

$$
\begin{array}{l}
\texttt{foo}\ (x,\ y,\ z)\ \{ \\
\quad \texttt{lock}_{\{y\}}\ x; \\
\quad \texttt{lock}_{\{z\}}\ y; \\
\quad \texttt{unlock}\ x; \\
\quad \texttt{lock}_{\emptyset}\ z; \\
\quad \texttt{unlock}\ z; \\
\quad \texttt{unlock}\ y \\
\}
\end{array}
$$

Boudol's Approach: Drawbacks

But, Boudol's approach is context insensitive and does not support unstructured locking.

$$
\begin{array}{l}
\texttt{foo}\ (x,\ y,\ z)\ \{ \\
\quad \texttt{lock}_{\{y\}}\ x; \\
\quad \texttt{lock}_{\{z\}}\ y; \\
\quad \texttt{unlock}\ x; \\
\quad \texttt{lock}_{\emptyset}\ z; \\
\quad \texttt{unlock}\ z; \\
\quad \texttt{unlock}\ y \\
\}
\end{array}
\qquad
\begin{array}{l}
\texttt{bar}\ (a,\ b)\ \{ \\
\quad \ldots \\
\quad \texttt{foo}\ (a,\ a,\ b) \\
\quad \ldots \\
\}
\end{array}
$$

Calling `foo` with x and y aliased.

Boudol's Approach: Drawbacks

But, Boudol's approach is context insensitive and does not support unstructured locking.

$\texttt{lock}_{\{a\}}\ a;$
$\texttt{lock}_{\{b\}}\ a;$
$\texttt{unlock}\ a;$
$\texttt{lock}_{\emptyset}\ b;$
$\texttt{unlock}\ b;$
$\texttt{unlock}\ a$

After substitution, the future locksets are wrong!

Boudol's Approach: Drawbacks

But, Boudol's approach is context insensitive and does not support unstructured locking.

$\texttt{lock}_{\{a\}}\ a;$ should be: $\{a, b\}$

$\texttt{lock}_{\{b\}}\ a;$ should be: $\emptyset$

$\texttt{unlock}\ a;$

$\texttt{lock}_{\emptyset}\ b;$

$\texttt{unlock}\ b;$

$\texttt{unlock}\ a$

After substitution, the future locksets are wrong!

Locking Patterns

Block Structured

```
bar (a, b) {
  lock a;
  lock b;
    ...
  unlock b;
  unlock a;
}
```

Locking Patterns

Block Structured

```
bar (a, b) {
  lock a;
  lock b;
     ...
  unlock b;
  unlock a;
}
```

Stack Based Same Function

```
bar (a) {
  lock a;
  if (...) {
    lock b;
    unlock b;
    unlock a;
    return; }
   ...
  unlock a;
  return; }
```

Locking Patterns

Block Structured

```
bar (a, b) {
  lock a;
  lock b;
    ...
  unlock b;
  unlock a;
}
```

Stack Based Same Function

```
bar (a) {
  lock a;
  if (...) {
    lock b;
    unlock b;
    unlock a;
    return; }
  ...
  unlock a;
  return; }
```

Stack Based Different Function

```
mylock (x) {
    lock x; }

bar (a) {
  mylock a;
  if (...) {
    unlock a;
  } return;
  ...
  unlock a;
  return; }
```

Locking Patterns

Block Structured

```
bar (a, b) {
  lock a;
  lock b;
    ...
  unlock b;
  unlock a;
}
```

Stack Based Same Function

```
bar (a) {
  lock a;
  if (...) {
    lock b;
    unlock b;
    unlock a;
    return; }
  ...
  unlock a;
  return; }
```

Stack Based Different Function

```
mylock (x) {
    lock x; }

bar (a) {
  mylock a;
  if (...) {
    unlock a;
  } return;
  ...
  unlock a;
  return; }
```

Unstructured Locking

```
bar (a, b) {
  lock a;
  lock b;
    ...
  unlock a;
  unlock b;
}
```

Do we really need support for unstructured locking?

Previous approaches:
issues with handling unstructured locking

Do we really need support for unstructured locking?

Previous approaches:
issues with handling unstructured locking

So, we gathered a codebase ($\sim$ 100 projects in C/Pthreads) and ran statistics on locking patterns.

Locking Pattern	Frequency
Block Structured	36.67%
Stack-Based (same function)	32.22%
Stack-Based (different function)	20.00%
Unstructured Locking	11.11%
Total	100.00%

Our Approach

Our implementation is based on the Type System designed by P. Gerakios *et al.*

Our Approach

Our implementation is based on the Type System designed by P. Gerakios *et al.*

Novelties compared to Boudol's system:

- Support for unstructured locking
- Support for inter-procedural effects

Our Approach

To support unstructured locking, we statically:

- track the order of `lock` and `unlock` operations
- annotate `lock` operations with a "continuation effect" (lock sequence of the code following that expression)

Our Approach

To support unstructured locking, we statically:

- track the order of `lock` and `unlock` operations
- annotate `lock` operations with a "continuation effect" (lock sequence of the code following that expression)

$$
\begin{array}{l}
\texttt{foo}\ (x,\ y,\ z)\ \{ \\
\quad \texttt{lock}_{[y+,\,x-,\,z+,\,z-,\,y-]}\ x; \\
\quad \texttt{lock}_{[x-,\,z+,\,z-,\,y-]}\ y; \\
\quad \texttt{unlock}\ x; \\
\quad \texttt{lock}_{[z-,\,y-]}\ z; \\
\quad \texttt{unlock}\ z; \\
\quad \texttt{unlock}\ y \\
\}
\end{array}
$$

Our Approach

To support unstructured locking, we statically:

- track the order of `lock` and `unlock` operations
- annotate `lock` operations with a "continuation effect" (lock sequence of the code following that expression)

```
foo (x, y, z) {
    lock[y+, x−, z+, z−, y−] x;
    lock[x−, z+, z−, y−] y;
    unlock x;
    lock[z−, y−] z;
    unlock z;
    unlock y
}
```

```
bar (a, b) {
    ...
    foo (a, a, b)
    ...
}
```

Our Approach

To support unstructured locking, we statically:

- track the order of `lock` and `unlock` operations
- annotate `lock` operations with a "continuation effect" (lock sequence of the code following that expression)

$\texttt{lock}_{[a+,\, a-,\, b+,\, b-,\, a-]}\ a;$
$\texttt{lock}_{[a-,\, b+,\, b-,\, a-]}\ a;$
$\texttt{unlock}\ x;$
$\texttt{lock}_{[b-,\, a-]}\ b;$
$\texttt{unlock}\ b;$
$\texttt{unlock}\ a$

Future locksets are then correctly calculated
After substitution, continuation effects are still valid!

Lockset Calculation

Compute future lockset at runtime using $lock$ annotations

Input: $\underbrace{a+}_{\text{lock operation}} \quad \underbrace{a+,\ a-,\ b+,\ b-,\ a-,\ \ldots}_{\text{continuation effect}}$

Lockset Calculation

Compute future lockset at runtime using $lock$ annotations

Input: $\underbrace{a+}_{\text{lock operation}} \quad \underbrace{a+,\ a-,\ b+,\ b-,\ a-,\ \ldots}_{\text{continuation effect}}$

- start with an empty future lockset

Lockset Calculation

Compute future lockset at runtime using $lock$ annotations

Input: $\underbrace{a+}_{\text{lock operation}} \quad \underbrace{a+,\ a-,\ b+,\ b-,\ a-,\ \ldots}_{\text{continuation effect}}$

- start with an empty future lockset
- traverse the continuation effect until the matching unlock operation (while there are more $a+$ than $a-$)

Lockset Calculation

Compute future lockset at runtime using $lock$ annotations

Input: $\underbrace{a+}_{\text{lock operation}} \quad \underbrace{a+,\ a-,\ b+,\ b-,\ a-,\ \ldots}_{\text{continuation effect}}$

- start with an empty future lockset
- traverse the continuation effect until the matching unlock operation (while there are more $a+$ than $a-$)
- add the locations being locked to the future lockset

Lockset Calculation

Compute future lockset at runtime using $lock$ annotations

Input: $\underbrace{a+}_{\text{lock operation}} \quad \underbrace{a+,\ a-,\ b+,\ b-,\ a-,\ \ldots}_{\text{continuation effect}}$

- start with an empty future lockset
- traverse the continuation effect until the matching unlock operation (while there are more $a+$ than $a-$)
- add the locations being locked to the future lockset

$$\text{lockset} = \{ \qquad \}$$

Lockset Calculation

Compute future lockset at runtime using $lock$ annotations

Input: $\underbrace{a+}_{\text{lock operation}} \quad \underbrace{a+,\ a-,\ b+,\ b-,\ a-,\ \ldots}_{\text{continuation effect}}$

- start with an empty future lockset
- traverse the continuation effect until the matching unlock operation (while there are more $a+$ than $a-$)
- add the locations being locked to the future lockset

$$\text{lockset} = \{\ a\ \}$$

Lockset Calculation

Compute future lockset at runtime using $lock$ annotations

Input: $\underbrace{a+}_{\text{lock operation}} \quad \underbrace{a+,\ a-,\ b+,\ b-,\ a-,\ \ldots}_{\text{continuation effect}}$

- start with an empty future lockset
- traverse the continuation effect until the matching unlock operation (while there are more $a+$ than $a-$)
- add the locations being locked to the future lockset

lockset $= \{\ a\ \ \ \}$

Lockset Calculation

Compute future lockset at runtime using $lock$ annotations

Input: $\underbrace{a+}_{\text{lock operation}} \quad \underbrace{a+,\ a-,\ \boxed{b+,}\ b-,\ a-,\ \ldots}_{\text{continuation effect}}$

- start with an empty future lockset
- traverse the continuation effect until the matching unlock operation (while there are more $a+$ than $a-$)
- add the locations being locked to the future lockset

$$\text{lockset} = \{\ a,\ \boxed{b}\ \}$$

Lockset Calculation

Compute future lockset at runtime using $lock$ annotations

Input: $\underbrace{a+}_{\text{lock operation}} \quad \underbrace{a+,\ a-,\ b+,\ \boxed{b-,}\ a-,\ \ldots}_{\text{continuation effect}}$

- start with an empty future lockset
- traverse the continuation effect until the matching unlock operation (while there are more $a+$ than $a-$)
- add the locations being locked to the future lockset

lockset $= \{\ a,\ \ b\ \}$

Lockset Calculation

Compute future lockset at runtime using $lock$ annotations

Input: $\underbrace{a+}_{\text{lock operation}} \quad \underbrace{a+,\ a-,\ b+,\ b-,\ \boxed{a-,}\ \ldots}_{\text{continuation effect}}$

- start with an empty future lockset
- traverse the continuation effect until the matching unlock operation (while there are more $a+$ than $a-$)
- add the locations being locked to the future lockset

lockset $= \{\ a,\ \ b\ \}$

Lockset Calculation

Compute future lockset at runtime using $lock$ annotations

Input: $\underbrace{a+}_{\text{lock operation}} \quad \underbrace{a+,\ a-,\ b+,\ b-,\ a-,\ \ldots}_{\text{continuation effect}}$

- start with an empty future lockset
- traverse the continuation effect until the matching unlock operation (while there are more $a+$ than $a-$)
- add the locations being locked to the future lockset

$$\text{lockset} = \{\ a,\ \ b\ \}$$

- but effects must not be intra-procedural !
- what happens if the matching unlock operation occurs after the function returns?

Inter-procedural Effects

- Function applications are also annotated with a "continuation effect"

Inter-procedural Effects

- Function applications are also annotated with a "continuation effect"
- When a function is applied, the continuation effect is pushed on a runtime stack

Inter-procedural Effects

- Function applications are also annotated with a "continuation effect"
- When a function is applied, the continuation effect is pushed on a runtime stack
- Lockset calculation may examine the stack

Inter-procedural Effects

- Function applications are also annotated with a "continuation effect"
- When a function is applied, the continuation effect is pushed on a runtime stack
- Lockset calculation may examine the stack

$$\texttt{foo}\;(\,)\{\quad \texttt{lock}_{[y+,\,y-]}\; x;\quad \texttt{lock}_{[y-]}\; y;\quad \texttt{unlock}\; y \quad \}$$

$$\texttt{bar}\;(\,)\;\{\quad \texttt{foo}()_{[z+]};\quad \texttt{lock}_{[]}\; z \quad \}$$

$$\texttt{main}\;(\,)\;\{\quad \texttt{bar}()_{[z-,\,x-]};\quad \texttt{unlock}\; z;\; \texttt{unlock}\; x \quad \}$$

Inter-procedural Effects

- Function applications are also annotated with a "continuation effect"
- When a function is applied, the continuation effect is pushed on a runtime stack
- Lockset calculation may examine the stack

$$\texttt{foo}\;(\,)\{\quad \texttt{lock}_{[y+,\,y-]}\;x;\quad \texttt{lock}_{[y-]}\;y;\quad \texttt{unlock}\;y\quad\}$$

$$\texttt{bar}\;(\,)\;\{\quad \texttt{foo}()_{[z+]};\quad \texttt{lock}_{[]}\;z\quad\}$$

$$\texttt{main}\;(\,)\;\{\quad \texttt{bar}()_{[z-,\,x-]};\quad \texttt{unlock}\;z;\;\texttt{unlock}\;x\quad\}$$

Inter-procedural Effects

- Function applications are also annotated with a "continuation effect"
- When a function is applied, the continuation effect is pushed on a runtime stack
- Lockset calculation may examine the stack

$\texttt{foo}\ (\,)\{\quad \texttt{lock}_{[y+,\,y-]}\ x;$

$\qquad\qquad \texttt{lock}_{[y-]}\ y;$

$\qquad\qquad \texttt{unlock}\ y \quad \}$

$\texttt{bar}\ (\,)\ \{\quad \texttt{foo}()_{[z+]};$

$\qquad\qquad \texttt{lock}_{[]}\ z \quad \}$

$\texttt{main}\ (\,)\ \{\quad \texttt{bar}()_{[z-,\,x-]};$

$\qquad\qquad \texttt{unlock}\ z;\ \texttt{unlock}\ x \quad \}$

Stack

$z-,\quad x-$

Inter-procedural Effects

- Function applications are also annotated with a "continuation effect"
- When a function is applied, the continuation effect is pushed on a runtime stack
- Lockset calculation may examine the stack

```
foo (){   lock[y+, y-] x;
          lock[y-] y;
          unlock y   }

bar ( ) {  foo()[z+];
           lock[] z   }

main ( ) {  bar()[z-, x-];
            unlock z; unlock x   }
```

Stack

$z-,\ \ x-$

Inter-procedural Effects

- Function applications are also annotated with a "continuation effect"
- When a function is applied, the continuation effect is pushed on a runtime stack
- Lockset calculation may examine the stack

$\texttt{foo}\,(\,)\{\ \ \texttt{lock}_{[y+,\,y-]}\ x;$
$\qquad\qquad\texttt{lock}_{[y-]}\ y;$
$\qquad\qquad\texttt{unlock}\ y\ \ \}$

$\texttt{bar}\,(\,)\ \{\ \ \texttt{foo}()_{[z+]};$
$\qquad\qquad\texttt{lock}_{[]}\ z\ \ \}$

$\texttt{main}\,(\,)\ \{\ \ \texttt{bar}()_{[z-,\,x-]};$
$\qquad\qquad\texttt{unlock}\ z;\ \texttt{unlock}\ x\ \ \}$

Stack

$z-,\ \ x-$
$z+$

Inter-procedural Effects

- Function applications are also annotated with a "continuation effect"
- When a function is applied, the continuation effect is pushed on a runtime stack
- Lockset calculation may examine the stack

$\texttt{foo}\ (\,)\{\ \ \texttt{lock}_{[y+,\,y-]}\ x;$
$\qquad\qquad \texttt{lock}_{[y-]}\ y;$
$\qquad\qquad \texttt{unlock}\ y\ \ \}$

$\texttt{bar}\ (\,)\ \{\ \ \texttt{foo}()_{[z+]};$
$\qquad\qquad \texttt{lock}_{[]}\ z\ \ \}$

$\texttt{main}\ (\,)\ \{\ \ \texttt{bar}()_{[z-,\,x-]};$
$\qquad\qquad \texttt{unlock}\ z;\ \texttt{unlock}\ x\ \ \}$

Stack

$z-,\ \ x-$
$z+$

Inter-procedural Effects

- Function applications are also annotated with a "continuation effect"
- When a function is applied, the continuation effect is pushed on a runtime stack
- Lockset calculation may examine the stack

$\texttt{foo}\;()\{\;\; \texttt{lock}_{[y+,\,y-]}\; x;$
$\qquad \texttt{lock}_{[y-]}\; y;$
$\qquad \texttt{unlock}\; y \;\; \}$

$\texttt{bar}\;()\;\{\;\; \texttt{foo}()_{[z+]};$
$\qquad \texttt{lock}_{[]}\; z \;\; \}$

$\texttt{main}\;()\;\{\;\; \texttt{bar}()_{[z-,\,x-]};$
$\qquad \texttt{unlock}\; z;\; \texttt{unlock}\; x \;\; \}$

Stack

$z-, \;\; x-$
$z+$

Lock/Continuation

$x+ \qquad y+, \;\; y-$

lockset = { }

Inter-procedural Effects

- Function applications are also annotated with a "continuation effect"
- When a function is applied, the continuation effect is pushed on a runtime stack
- Lockset calculation may examine the stack

foo ( ){ $\texttt{lock}_{[y+,\,y-]}\ x$;
$\texttt{lock}_{[y-]}\ y$;
$\texttt{unlock}\ y$ }

bar ( ) { $\texttt{foo}()_{[z+]}$;
$\texttt{lock}_{[]}\ z$ }

main ( ) { $\texttt{bar}()_{[z-,\,x-]}$;
$\texttt{unlock}\ z$; $\texttt{unlock}\ x$ }

Stack

$z-,\ \ x-$
$z+$

Lock/Continuation

$x+$ $y+,$ $y-$

lockset = { y }

Inter-procedural Effects

- Function applications are also annotated with a "continuation effect"
- When a function is applied, the continuation effect is pushed on a runtime stack
- Lockset calculation may examine the stack

$\texttt{foo}\ (\,)\{\ \ \texttt{lock}_{[y+,\,y-]}\ x;$
$\texttt{lock}_{[y-]}\ y;$
$\texttt{unlock}\ y\ \ \}$

$\texttt{bar}\ (\,)\ \{\ \ \texttt{foo}()_{[z+]};$
$\texttt{lock}_{[]}\ z\ \ \}$

$\texttt{main}\ (\,)\ \{\ \ \texttt{bar}()_{[z-,\,x-]};$
$\texttt{unlock}\ z;\ \texttt{unlock}\ x\ \ \}$

Stack

$z-,\ \ x-$
$z+$

Lock/Continuation

$x+\quad y+,\ \ y-$

lockset = { y }

Inter-procedural Effects

- Function applications are also annotated with a "continuation effect"
- When a function is applied, the continuation effect is pushed on a runtime stack
- Lockset calculation may examine the stack

$\texttt{foo}\ (\,)\{\ \ \texttt{lock}_{[y+,\,y-]}\ x;$
$\qquad\texttt{lock}_{[y-]}\ y;$
$\qquad\texttt{unlock}\ y\ \ \}$

$\texttt{bar}\ (\,)\ \{\ \ \texttt{foo}()_{[z+]};$
$\qquad\texttt{lock}_{[]}\ z\ \ \}$

$\texttt{main}\ (\,)\ \{\ \ \texttt{bar}()_{[z-,\,x-]};$
$\qquad\texttt{unlock}\ z;\ \texttt{unlock}\ x\ \ \}$

Stack

$z-,\ \ x-$
$z+$

Lock/Continuation

$x+\quad y+,\ \ y-$

lockset = { $y,\ \ z$ }

Inter-procedural Effects

- Function applications are also annotated with a "continuation effect"
- When a function is applied, the continuation effect is pushed on a runtime stack
- Lockset calculation may examine the stack

$\texttt{foo}\ (\,)\{\ \ \texttt{lock}_{[y+,\,y-]}\ x;$
$\quad\texttt{lock}_{[y-]}\ y;$
$\quad\texttt{unlock}\ y\ \ \}$

$\texttt{bar}\ (\,)\ \{\ \ \texttt{foo}()_{[z+]};$
$\quad\texttt{lock}_{[]}\ z\ \ \}$

$\texttt{main}\ (\,)\ \{\ \ \texttt{bar}()_{[z-,\,x-]};$
$\quad\texttt{unlock}\ z;\ \texttt{unlock}\ x\ \ \}$

Stack

$z-,\ \ x-$
$z+$

Lock/Continuation

$x+\quad y+,\ \ y-$

lockset = $\{\ y,\ \ z\ \}$

Inter-procedural Effects

- Function applications are also annotated with a "continuation effect"
- When a function is applied, the continuation effect is pushed on a runtime stack
- Lockset calculation may examine the stack

```
foo (){   lock[y+, y−] x;
          lock[y−] y;
          unlock y   }

bar () {   foo()[z+];
           lock[] z   }

main () {   bar()[z−, x−];
            unlock z; unlock x   }
```

Stack

$z-,\ x-$
$z+$

Lock/Continuation

$x+ \quad y+,\ y-$

lockset = $\{\ y,\ \ z\ \}$

Conditional Expressions

$\boxed{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2}$

- How can we type-check conditionals ?

Conditional Expressions

$\boxed{\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2}$

- How can we type-check conditionals ?
- Consider:

$\texttt{if } e \texttt{ then } (\texttt{lock}_{[]}\ x; \ldots \texttt{unlock } x)$ effect: $x+$, $x-$

$\quad \texttt{else skip}$ effect: empty

Conditional Expressions

$\boxed{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2}$

- How can we type-check conditionals ?
- Consider:

  $\texttt{if}\ e\ \texttt{then}\ (\texttt{lock}_{[]}\ x; \ldots \texttt{unlock}\ x)$ effect: $x+$, $x-$

  $\quad\texttt{else skip}$ effect: empty

- Conservative, require: $\texttt{effect}(e_1) = \texttt{effect}(e_2)$

Conditional Expressions

$\boxed{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2}$

- How can we type-check conditionals ?
- Consider:

$\texttt{if}\ e\ \texttt{then}\ (\texttt{lock}_{[]}\ x; \ldots \texttt{unlock}\ x)$ effect: $x+,\ x-$

$\quad\quad \texttt{else skip}$ effect: empty

- Conservative, require: $\texttt{effect}(e_1) = \texttt{effect}(e_2)$
- We require: $\texttt{overall}(\texttt{effect}(e_1)) = \texttt{overall}(\texttt{effect}(e_2))$
- The conditional has effect: $\texttt{effect}(e_1)\ ?\ \texttt{effect}(e_2)$

Conditional Expressions

$\boxed{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2}$

- How can we type-check conditionals ?
- Consider:

$\texttt{if}\ e\ \texttt{then}\ (\texttt{lock}_{[]}\ x; \ldots \texttt{unlock}\ x)$ effect: $x+, x-$

$\quad\texttt{else skip}$ effect: empty

- Conservative, require: $\texttt{effect}(e_1) = \texttt{effect}(e_2)$
- We require: $\texttt{overall}(\texttt{effect}(e_1)) = \texttt{overall}(\texttt{effect}(e_2))$
- The conditional has effect: $\texttt{effect}(e_1)\ ?\ \texttt{effect}(e_2)$
- Runtime lockset calculation for conditionals:

"future lockset"$((\gamma_1\ ?\ \gamma_2); \gamma) =$ "future lockset"$(\gamma_1; \gamma) \cup$ "future lockset"$(\gamma_2; \gamma)$

Loop Expressions

Loop Expressions

$\boxed{\texttt{while}\ e\ \texttt{then}\ e_1}$

Consider:

```
lock x;
while (...) {
  lock y;
  unlock y;
  unlock x;
   ...
  lock x;
}
unlock x
```

Loop Expressions

$\boxed{\texttt{while}\ e\ \texttt{then}\ e_1}$

Consider:

$\texttt{lock}\ x;$
$\texttt{while}\ (\ldots)\ \{$
$\quad \texttt{lock}\ y;$
$\quad \texttt{unlock}\ y;$
$\quad \texttt{unlock}\ x;$
$\quad \ldots$
$\quad \texttt{lock}\ x;$
$\}$
$\texttt{unlock}\ x$

Loop Expressions

$\boxed{\texttt{while}\ e\ \texttt{then}\ e_1}$

Consider:

$\texttt{lock}\ x;$
$\texttt{while}\ (\ldots)\ \{$
  $\texttt{lock}\ y;$
  $\texttt{unlock}\ y;$
  $\texttt{unlock}\ x;$
  $\ldots$
  $\texttt{lock}\ x;$
$\}$
$\texttt{unlock}\ x$

Loop Expressions

$\boxed{\texttt{while}\ e\ \texttt{then}\ e_1}$

Consider:

$\texttt{lock}\ x;$
$\texttt{while}\ (\ldots)\ \{$
  $\texttt{lock}\ y;$
  $\texttt{unlock}\ y;$
  $\texttt{unlock}\ x;$
    $\ldots$
  $\texttt{lock}\ x;$
$\}$
$\texttt{unlock}\ x$

Loop Expressions

$\boxed{\texttt{while}\ e\ \texttt{then}\ e_1}$

- Conservative, require:

  $\texttt{effect}(e_1) = \emptyset$

Consider:

$\texttt{lock}\ x;$
$\texttt{while}\ (\ldots)\ \{$
$\quad\texttt{lock}\ y;$
$\quad\texttt{unlock}\ y;$
$\quad\texttt{unlock}\ x;$
$\quad\ \ldots$
$\quad\texttt{lock}\ x;$
$\}$
$\texttt{unlock}\ x$

Loop Expressions

$\boxed{\texttt{while}\ e\ \texttt{then}\ e_1}$

Consider:

$\texttt{lock}\ x;$
$\texttt{while}\ (\ldots)\ \{$
$\quad \texttt{lock}\ y;$
$\quad \texttt{unlock}\ y;$
$\quad \texttt{unlock}\ x;$
$\quad \ldots$
$\quad \texttt{lock}\ x;$
$\}$
$\texttt{unlock}\ x$

- Conservative, require:
  $\texttt{effect}(e_1) = \emptyset$
- We, require:
  $\texttt{overall}(\texttt{effect}(before\ loop)) = \texttt{overall}(\texttt{effect}(after\ loop))$

Loop Expressions

$\boxed{\texttt{while}\ e\ \texttt{then}\ e_1}$

Consider:

$\texttt{lock}\ x;$
$\texttt{while}\ (\ldots)\ \{$
$\quad\texttt{lock}\ y;$
$\quad\texttt{unlock}\ y;$
$\quad\texttt{unlock}\ x;$
$\quad\ldots$
$\quad\texttt{lock}\ x;$
$\}$
$\texttt{unlock}\ x$

- Conservative, require:
  $\texttt{effect}(e_1) = \emptyset$
- We, require:
  $\texttt{overall}(\texttt{effect}(before\ loop)) = \texttt{overall}(\texttt{effect}(after\ loop))$
- The whole construct has effect:
  $[\,]\ ?\ (\texttt{effect}(e_1);\ \underbrace{([\,]\ ?\ \texttt{summary}(\texttt{effect}(e_1))))}_{\text{handles successive iterations}}$
- Summary: multiple lock/unlock pairs are redundant

Loop Expressions

$\boxed{\texttt{while}\ e\ \texttt{then}\ e_1}$

Consider:

$\texttt{lock}\ x;$
$\texttt{while}\ (\ldots)\ \{$
$\quad\texttt{lock}\ y;$
$\quad\texttt{unlock}\ y;$
$\quad\texttt{unlock}\ x;$
$\quad\ldots$
$\quad\texttt{lock}\ x;$
$\}$
$\texttt{unlock}\ x$

- Conservative, require:
  $\texttt{effect}(e_1) = \emptyset$
- We, require:
  $\texttt{overall}(\texttt{effect}(before\ loop)) = \texttt{overall}(\texttt{effect}(after\ loop))$
- The whole construct has effect:
  $[\,]\,?\,(\texttt{effect}(e_1);\ \underbrace{([\,]\,?\,\texttt{summary}(\texttt{effect}(e_1))))}_{\text{handles successive iterations}}$
- Summary: multiple lock/unlock pairs are redundant
- Runtime lockset calculation: as in conditional expressions

Locking Algorithm

Key Idea:
A lock operation succeeds only when both the lock and its future lockset are available.

Locking Algorithm

Key Idea:
A lock operation succeeds only when both the lock and its future lockset are available.

Algorithm:

1. Compute the future lockset.
2. Check if the future lockset is available.
3. If it is, then acquire the mutex tentatively,
   otherwise wait shortly and go back to step 2.
4. Check (again) if the future lockset is available.
5. If it is, then return successfully,
   otherwise unlock and go back to step 2.

Locking Algorithm

Key Idea:
A lock operation succeeds only when both the lock and its future lockset are available.

Algorithm:

1. Compute the future lockset.
2. Check if the future lockset is available.
3. If it is, then acquire the mutex tentatively,
   otherwise wait shortly and go back to step 2.
4. Check (again) if the future lockset is available.
5. If it is, then return successfully,
   otherwise unlock and go back to step 2.

Locking Algorithm

Key Idea:
A lock operation succeeds only when both the lock and its future lockset are available.

Algorithm:

1. Compute the future lockset.
2. Check if the future lockset is available.
3. If it is, then acquire the mutex tentatively,
   otherwise wait shortly and go back to step 2.
4. Check (again) if the future lockset is available.
5. If it is, then return successfully,
   otherwise unlock and go back to step 2.

Locking Algorithm

Key Idea:
A lock operation succeeds only when both the lock and its future lockset are available.

Algorithm:

1. Compute the future lockset.
2. Check if the future lockset is available.
3. If it is, then acquire the mutex tentatively,
   otherwise wait shortly and go back to step 2.
4. Check (again) if the future lockset is available.
5. If it is, then return successfully,
   otherwise unlock and go back to step 2.

Locking Algorithm

Key Idea:
A lock operation succeeds only when both the lock and its future lockset are available.

Algorithm:

1. Compute the future lockset.
2. Check if the future lockset is available.
3. If it is, then acquire the mutex tentatively, otherwise wait shortly and go back to step 2.
4. Check (again) if the future lockset is available.
5. If it is, then return successfully, otherwise unlock and go back to step 2.

Errrr...

Errrr...

- Q: Why does this work?

Errrr...

- Q: Why does this work?
  A: Tackle 4th condition: circular wait

Errrr...

- Q: Why does this work?
  A: Tackle 4th condition: circular wait
- Q: Why do we compute the future lockset at runtime?

Errrr...

- Q: Why does this work?
  A: Tackle 4th condition: circular wait

- Q: Why do we compute the future lockset at runtime?
  A: We need info from runtime stack – don't have context sensitive analyis

Errrr. . .

- Q: Why does this work?
  A: Tackle 4th condition: circular wait
- Q: Why do we compute the future lockset at runtime?
  A: We need info from runtime stack – don´t have context sensitive analyis
- Q: Why not acquire the future lockset preemptively?

Errrr. . .

- Q: Why does this work?
  A: Tackle 4^{th} condition: circular wait
- Q: Why do we compute the future lockset at runtime?
  A: We need info from runtime stack – don´t have context sensitive analyis
- Q: Why not acquire the future lockset preemptively?
  A: Limits degree of parallelism.

Errrr. . .

- Q: Why does this work?
  A: Tackle 4th condition: circular wait
- Q: Why do we compute the future lockset at runtime?
  A: We need info from runtime stack – don't have context sensitive analyis
- Q: Why not acquire the future lockset preemptively?
  A: Limits degree of parallelism.
- Q: Don't we have to apply the locking algorithm atomically?

Errrr...

- Q: Why does this work?
  A: Tackle 4th condition: circular wait
- Q: Why do we compute the future lockset at runtime?
  A: We need info from runtime stack – don't have context sensitive analyis
- Q: Why not acquire the future lockset preemptively?
  A: Limits degree of parallelism.
- Q: Don't we have to apply the locking algorithm atomically?
  A: Nope, it suffices that the future lockset is available momentarily before acquiring the lock.

Analysis Background

Target language: C/Pthreads

Analyzing C language: Not so easy . . .

Analysis Background

Target language: C/Pthreads

Analyzing C language: Not so easy . . .

- CIL

- RELAY

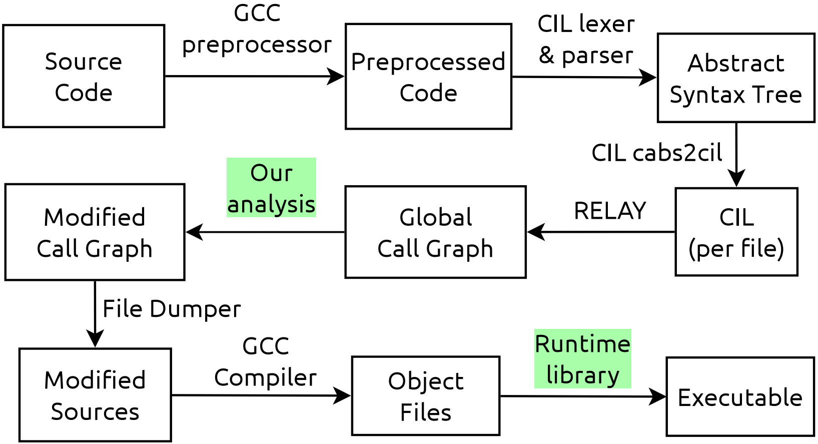
Analysis Background

Target language: C/Pthreads

Analyzing C language: Not so easy . . .

- CIL
  - Parse C, simplified intermediate language
  - Control flow graph
  - Dataflow framework
  - Function pointer analysis
- RELAY
  - Global Call Graph
  - Pointer analysis

Workflow


Source Code
GCC preprocessor
Preprocessed Code
CIL lexer & parser
Abstract Syntax Tree
CIL cabs2cil
CIL (per file)
RELAY
Global Call Graph
Our analysis
Modified Call Graph
File Dumper
Modified Sources
GCC Compiler
Object Files
Runtime library
Executable

Tool Overview

- Static Analysis:
  - Traverses the call graph Bottom-up:
    express everything modulo globals
    and formals.
  - Infers effects for each function
  - Instruments code with dynamic effects

Tool Overview

- Static Analysis:
  - Traverses the call graph Bottom-up:
    express everything modulo globals
    and formals.
  - Infers effects for each function
  - Instruments code with dynamic effects
- Runtime System:
  - Overrides Pthread library
  - Uses dynamic effects to avoid deadlocks

Analyzing functions

- Symbolic Execution (module from RELAY):
  - Keeps state for every program point
  - Uses flow information
  - Computes symbolic map:
    lvalues -> global, formal values
  - Our contribution:
    - track visible writes at function calls
    - heap allocation support

Analyzing functions

- Symbolic Execution (module from RELAY):
  - Keeps state for every program point
  - Uses flow information
  - Computes symbolic map:
    lvalues -> global, formal values
  - Our contribution:
    - track visible writes at function calls
    - heap allocation support

- Effect inference

Analyzing functions

- Symbolic Execution (module from RELAY):
  - Keeps state for every program point
  - Uses flow information
  - Computes symbolic map:
    lvalues -> global, formal values
  - Our contribution:
    - track visible writes at function calls
    - heap allocation support
- Effect inference
- Effect optimizations

Analyzing functions

- Symbolic Execution (module from RELAY):
  - Keeps state for every program point
  - Uses flow information
  - Computes symbolic map:
    lvalues -> global, formal values
  - Our contribution:
    - track visible writes at function calls
    - heap allocation support
- Effect inference
- Effect optimizations
- Loop effects (special treatement)

Analyzing functions

- Symbolic Execution (module from RELAY):
  - Keeps state for every program point
  - Uses flow information
  - Computes symbolic map:
    lvalues -> global, formal values
  - Our contribution:
    - track visible writes at function calls
    - heap allocation support
- Effect inference
- Effect optimizations
- Loop effects (special treatement)

Effect Description

List of atomic operations:

Effect Description

List of atomic operations:

- Lock/Unlock operation ($l+$ or $l-$)

Effect Description

List of atomic operations:

- Lock/Unlock operation ($l+$ or $l-$)
- Joint effect ($\gamma_1 \, ? \, \cdots \, ? \, \gamma_n$):
  Multiple possible paths of execution $\Rightarrow$ alternate effects

Effect Description

List of atomic operations:

- Lock/Unlock operation ($l+$ or $l-$)
- Joint effect ($\gamma_1 ? \cdots ? \gamma_n$):
  Multiple possible paths of execution $\Rightarrow$ alternate effects
- Function call:
  - Compute summary of the callee's effect
  - Substitute it to the caller's context
  - Inline it at call site
  - Function pointers: joint effect of all possible targets

Effect Description

List of atomic operations:

- Lock/Unlock operation ($l+$ or $l-$)
- Joint effect ($\gamma_1 \,?\, \cdots \,?\, \gamma_n$):
  Multiple possible paths of execution $\Rightarrow$ alternate effects
- Function call:
  - Compute summary of the callee's effect
  - Substitute it to the caller's context
  - Inline it at call site
  - Function pointers: joint effect of all possible targets
- Loop placeholder:
  Backedges in CFG $\Rightarrow$
  Use placeholder and backpatch with loop effect later

Effect Description

List of atomic operations:

- Lock/Unlock operation ($l+$ or $l-$)
- Joint effect ($\gamma_1 ? \cdots ? \gamma_n$):
  Multiple possible paths of execution $\Rightarrow$ alternate effects
- Function call:
  - Compute summary of the callee's effect
  - Substitute it to the caller's context
  - Inline it at call site
  - Function pointers: joint effect of all possible targets
- Loop placeholder:
  Backedges in CFG $\Rightarrow$
  Use placeholder and backpatch with loop effect later

Challenge: create a linear effect from a complex CFG.

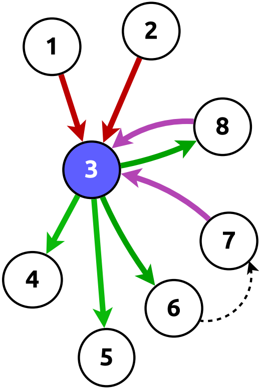
Effect Inference

Uses CIL's dataflow framework.

Effect Inference

Uses CIL's dataflow framework.

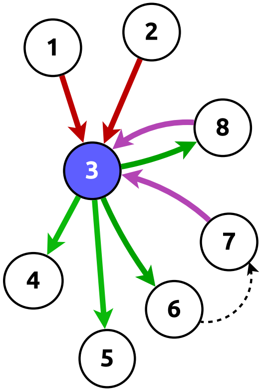
- State: input, current, output, loop effect



Effect Inference

Uses CIL's dataflow framework.

- State: input, current, output, loop effect
- Initialization of current effect:
  - Compute node/statement effects
  - Placeholders for loops



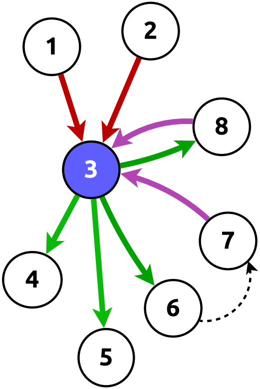
Effect Inference

Uses CIL's dataflow framework.

- State: input, current, output, loop effect
- Initialization of current effect:
  - Compute node/statement effects
  - Placeholders for loops
- Combine predecessors:

  output effect =
   join(input effects) ; current

  loop effect =
   join(incomming loop effects)



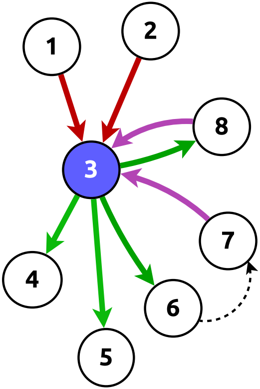
Effect Inference

Uses CIL's dataflow framework.

- State: input, current, output, loop effect
- Initialization of current effect:
  - Compute node/statement effects
  - Placeholders for loops
- Combine predecessors:

  output effect =
  join(input effects) ; current

  loop effect =
  join(incomming loop effects)
- Function's effect: joint effect at return statements



Post-Computation Effect Operations

- Backpatch loop placeholders with loop effects

Post-Computation Effect Operations

- Backpatch loop placeholders with loop effects
- Optimizations:

Post-Computation Effect Operations

- Backpatch loop placeholders with loop effects
- Optimizations:
  - Common prefix/suffix

    $([A, B, C] \mathbin{?} [A, B, D] \mathbin{?} [E, F, G] \mathbin{?} [E, H, I]) \rightarrow$

    $(([A, B]; ([C] \mathbin{?} [D])) \mathbin{?} ([E]; ([F, G] \mathbin{?} [H, I])))$

Post-Computation Effect Operations

- Backpatch loop placeholders with loop effects
- Optimizations:
  - Common prefix/suffix

    $([A, B, C] \mathbin{?} [A, B, D] \mathbin{?} [E, F, G] \mathbin{?} [E, H, I]) \rightarrow$

    $(([A, B]; ([C] \mathbin{?} [D])) \mathbin{?} ([E]; ([F, G] \mathbin{?} [H, I])))$

Post-Computation Effect Operations

- Backpatch loop placeholders with loop effects
- Optimizations:
  - Common prefix/suffix

$$([A, B, C] \mathbin{?} [A, B, D] \mathbin{?} [E, F, G] \mathbin{?} [E, H, I]) \rightarrow$$
$$(([A, B]; ([C] \mathbin{?} [D])) \mathbin{?} ([E]; ([F, G] \mathbin{?} [H, I])))$$

Post-Computation Effect Operations

- Backpatch loop placeholders with loop effects
- Optimizations:
  - Common prefix/suffix
    $([A, B, C] \mathbin{?} [A, B, D] \mathbin{?} [E, F, G] \mathbin{?} [E, H, I]) \rightarrow$
    $(([A, B]; ([C] \mathbin{?} [D])) \mathbin{?} ([E]; ([F, G] \mathbin{?} [H, I])))$
  - Flatten effects
    $((\gamma_1 \mathbin{?} \gamma_2) \mathbin{?} (\gamma_3 \mathbin{?} \gamma_4)) \rightarrow (\gamma_1 \mathbin{?} \gamma_2 \mathbin{?} \gamma_3 \mathbin{?} \gamma_4)$

Post-Computation Effect Operations

- Backpatch loop placeholders with loop effects
- Optimizations:
  - Common prefix/suffix
    $([A,B,C] ? [A,B,D] ? [E,F,G] ? [E,H,I]) \rightarrow$
    $(([A,B];([C] ? [D])) ? ([E];([F,G] ? [H,I])))$
  - Flatten effects
    $((\gamma_1 ? \gamma_2) ? (\gamma_3 ? \gamma_4)) \rightarrow (\gamma_1 ? \gamma_2 ? \gamma_3 ? \gamma_4)$
  - Remove redundant empty effects
    $(\gamma_1 ? [\,] ? \gamma_2 ? [\,]) \rightarrow (\gamma_1 ? [\,] ? \gamma_2)$
    $(\gamma ? [\,]) \rightarrow \gamma$ when γ has no unmatched operations

Post-Computation Effect Operations

- Backpatch loop placeholders with loop effects
- Optimizations:
  - Common prefix/suffix
    $([A, B, C] \mathbin{?} [A, B, D] \mathbin{?} [E, F, G] \mathbin{?} [E, H, I]) \rightarrow$
    $(([A, B]; ([C] \mathbin{?} [D])) \mathbin{?} ([E]; ([F, G] \mathbin{?} [H, I])))$
  - Flatten effects
    $((\gamma_1 \mathbin{?} \gamma_2) \mathbin{?} (\gamma_3 \mathbin{?} \gamma_4)) \rightarrow (\gamma_1 \mathbin{?} \gamma_2 \mathbin{?} \gamma_3 \mathbin{?} \gamma_4)$
  - Remove redundant empty effects
    $(\gamma_1 \mathbin{?} [\,] \mathbin{?} \gamma_2 \mathbin{?} [\,]) \rightarrow (\gamma_1 \mathbin{?} [\,] \mathbin{?} \gamma_2)$
    $(\gamma \mathbin{?} [\,]) \rightarrow \gamma$ when γ has no unmatched operations
  - Run the above alternately to a fixed point.

Code generation

Code generation

- Goal: annotate every lock operation/function call with effect frame inducing a minimal overhead

Code generation

- Goal: annotate every lock operation/function call with effect frame inducing a minimal overhead
- Single block of initialization code for each function
- Effect index update instructions before each call and lock operation.

Code generation

- Goal: annotate every lock operation/function call with effect frame inducing a minimal overhead
- Single block of initialization code for each function
- Effect index update instructions before each call and lock operation.
- Each function: instructions for pushing and popping effects from the runtime stack at function entry and exit points

Code generation

- Goal: annotate every lock operation/function call with effect frame inducing a minimal overhead
- Single block of initialization code for each function
- Effect index update instructions before each call and lock operation.
- Each function: instructions for pushing and popping effects from the runtime stack at function entry and exit points
- Mappings for stack and heap references

Code generation

- Goal: annotate every lock operation/function call with effect frame inducing a minimal overhead
- Single block of initialization code for each function
- Effect index update instructions before each call and lock operation.
- Each function: instructions for pushing and popping effects from the runtime stack at function entry and exit points
- Mappings for stack and heap references

$\Rightarrow$ These impose a constant overhead

Stack references

- Enumeration on used stack addresses
- Create array with addresses
- Access address throught index

Stack references

- Enumeration on used stack addresses
- Create array with addresses
- Access address throught index

Original version

```
struct bar {
  lock_t a;
  lock_t b;
};

foo(struct bar * s,
    lock_t * p) {
  lock( & s->a );
  lock( & s->b );
  lock( p );
}
```

Stack references

- Enumeration on used stack addresses
- Create array with addresses
- Access address throught index

Original version

```
struct bar {
  lock_t a;
  lock_t b;
};

foo(struct bar * s,
    lock_t * p) {
  lock( & s->a );
  lock( & s->b );
  lock( p );
}
```

Modified version

```
foo(struct bar *s, mylock_t *p)
{
  __cil_tmp5.locals = __cil_tmp4;

  __cil_tmp4[0] = (mylock *)(& s->a);
  __cil_tmp4[1] = (mylock_t *)(& s->b);
  __cil_tmp4[2] = (mylock_t *)p;
  ...
}
```

Heap Allocation

```
lock_t * foo() {
  return malloc(); //Loc A
}

bar(){
  lock_t * a = foo(); //Loc B
  lock(a);

  lock_t * b = foo(); //Loc C
  lock(b);
}

main() {
  bar();     //Loc D

  bar();     //Loc E
}
```

Heap Allocation

```
lock_t * foo() {
  return malloc(); //Loc A
}

bar(){
  lock_t * a = foo(); //Loc B
  lock(a);

  lock_t * b = foo(); //Loc C
  lock(b);
}

main() {
  bar();    //Loc D

  bar();    //Loc E
}
```

- Heap model: based on the context of allocating function

Heap Allocation

```
lock_t * foo() {
  return malloc(); //Loc A
}

bar(){
  lock_t * a = foo(); //Loc B
  lock(a);

  lock_t * b = foo(); //Loc C
  lock(b);
}

main() {
  bar();    //Loc D

  bar();    //Loc E
}
```

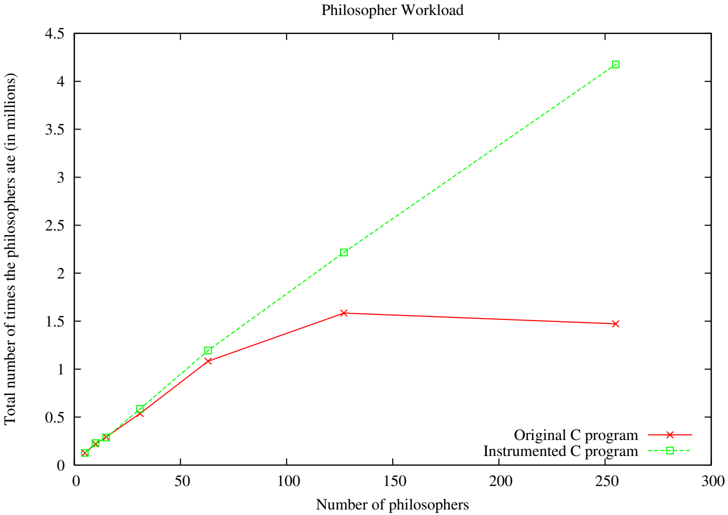
- Heap model: based on the context of allocating function
- Distinct abstract locations:
  $(Loc\ D, Loc\ B, Loc\ A)$
  $(Loc\ D, Loc\ C, Loc\ A)$
  $(Loc\ E, Loc\ B, Loc\ A)$
  $(Loc\ E, Loc\ C, Loc\ A)$

Heap Allocation

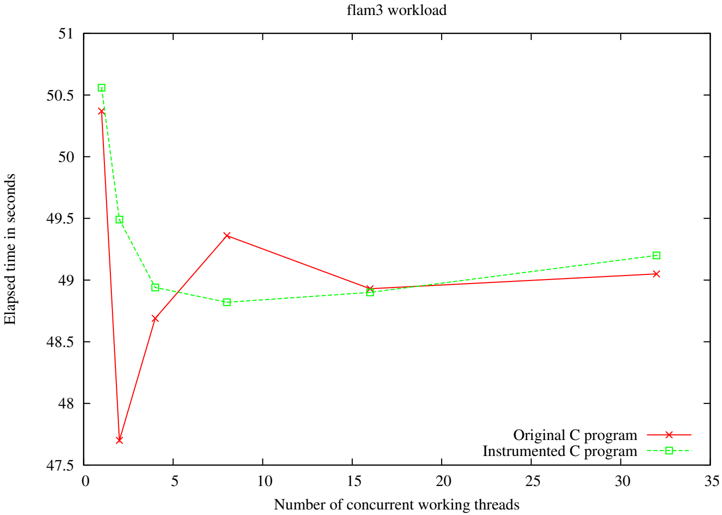
```
lock_t * foo() {
  return malloc(); //Loc A
}

bar(){
  lock_t * a = foo(); //Loc B
  lock(a);

  lock_t * b = foo(); //Loc C
  lock(b);
}

main() {
  bar();    //Loc D

  bar();    //Loc E
}
```

- Heap model: based on the context of allocating function
- Distinct abstract locations:
  $(Loc\ D, Loc\ B, Loc\ A)$
  $(Loc\ D, Loc\ C, Loc\ A)$
  $(Loc\ E, Loc\ B, Loc\ A)$
  $(Loc\ E, Loc\ C, Loc\ A)$
- Dynamic mapping (hashtable):
  abstract location $\leftrightarrow$
  runtime address

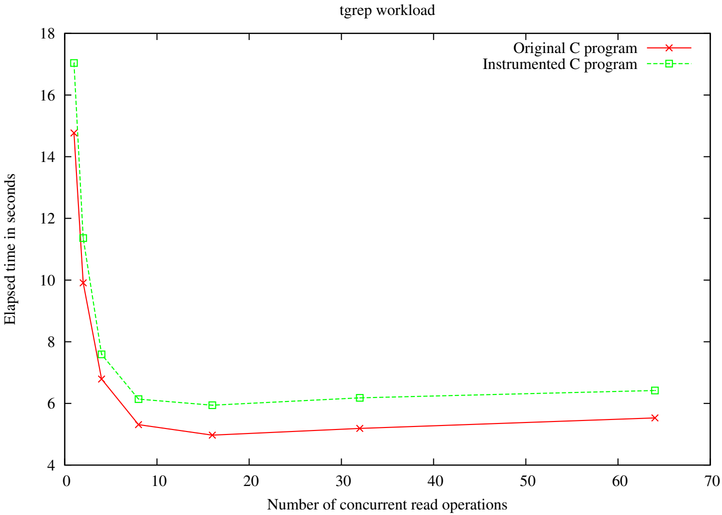
Evaluation: Dining Philosophers



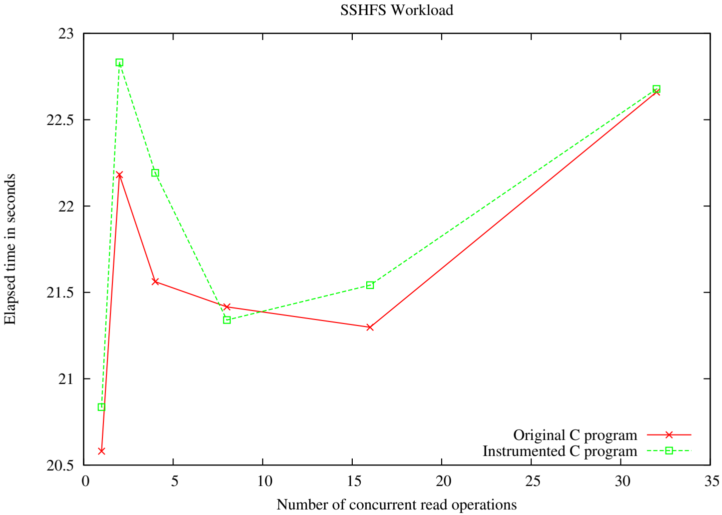
Evaluation: Cosmic Fractal Frames



Evaluation: Multi-threaded "grep"



Evaluation: File System over SSH



Limitations

- Non C code, assembly
- Pointer analysis: recursive structs, pointer arithmetic
- Heap allocation in loops, recursive functions

Limitations

- Non C code, assembly
- Pointer analysis: recursive structs, pointer arithmetic
- Heap allocation in loops, recursive functions

Future Work:

- Accept annotations for library code
- More precise pointer analysis, higher level of indirection
- Recursive functions (supported by type system)
- Static locksets, caching

Thank you!

Questions?