Dynamic Deadlock Avoidance
Using Statically Inferred Effects

Kostis Sagonas\textsuperscript{1,2}

joint work with

P. Gerakios\textsuperscript{1}

N. Papaspyrou\textsuperscript{1}

P. Vekris\textsuperscript{1,3}

\textsuperscript{1} School of ECE, National Technical University of Athens, Greece
\textsuperscript{2} Dept. of Information Technology, Uppsala University, Sweden
\textsuperscript{3} Dept. of Computer Science, UC San Diego, U.S.A.
Long-term Research Goal

Enhance reliability of concurrent systems software by designing and implementing low-level languages with static guarantees for absence of certain errors.
Long-term Research Goal

Enhance reliability of concurrent systems software by designing and implementing low-level languages with static guarantees for absence of certain errors.

Prior work:
- safe multithreading in a language with shared-memory and a common hierarchy of regions and locks
- memory safety and race freedom
- implemented in an extended Cyclone
Long-term Research Goal

Enhance reliability of concurrent systems software by designing and implementing low-level languages with static guarantees for absence of certain errors.

Prior work:

- safe multithreading in a language with shared-memory and a common hierarchy of regions and locks
- memory safety and race freedom
- implemented in an extended Cyclone

Safety properties ...
Long-term Research Goal

Enhance reliability of concurrent systems software by designing and implementing low-level languages with static guarantees for absence of certain errors.

Prior work:
- safe multithreading in a language with shared-memory and a common hierarchy of regions and locks
- memory safety and race freedom
- implemented in an extended Cyclone

Safety properties... liveness?
This Talk is About Deadlock Avoidance

In a low-level language suitable for systems programming
  ▶ at the C level of abstraction
  ▶ unstructured locking primitives (lock/unlock)
This Talk is About Deadlock Avoidance

In a low-level language suitable for systems programming

» at the C level of abstraction
» unstructured locking primitives (lock/unlock)

Tool for C/pthreads programs

» with a static analysis component that annotates programs with continuation effects of locks and
» links them with a runtime system (pthread library replacement) that knows how to avoid deadlocks
This Talk is About Deadlock Avoidance

In a low-level language suitable for systems programming

- at the C level of abstraction
- unstructured locking primitives (lock/unlock)

Tool for C/pthreads programs

- with a static analysis component that annotates programs with continuation effects of locks and
- links them with a runtime system (pthread library replacement) that knows how to avoid deadlocks

Evaluation results
What is a Deadlock?

- two or more threads form a circular chain
- each thread waits for a lock held by the next thread in chain
Approaches to Deadlock Freedom

Prevention

“correct by design”
Approaches to Deadlock Freedom

Prevention

“correct by design”

Detection and recovery

transactional semantics
Approaches to Deadlock Freedom

Prevention

“correct by design”

Detection and recovery

transactional semantics

Avoidance

predict possible deadlock
Deadlock Prevention: A Static Approach

Key idea:
- impose a single global lock order
- check that all threads respect this lock order
Deadlock Prevention: A Static Approach

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: A Static Approach

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

However, a global lock order is restrictive:
\[
\{ \text{lock}(x); \ldots \text{lock}(y); \ldots \} \ || \ \{ \text{lock}(y); \ldots \text{lock}(x); \ldots \} 
\]
Deadlock Prevention: A Static Approach

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

However, a global lock order is restrictive:

\[
\begin{align*}
\{ \text{lock}(x); \ldots \text{lock}(y); \ldots \} & \parallel \{ \text{lock}(y); \ldots \text{lock}(x); \ldots \} \\
& \quad \text{subject to } x \leq y \quad \text{and} \quad y \leq x
\end{align*}
\]

- no single global order $\Rightarrow$ reject program
Deadlock Avoidance: A Hybrid Approach

Basic idea:

- **statically**: for each lock operation compute information that will allow the computation of its “future lockset”
- **dynamically**: during runtime check that the “future lockset” is available before granting the lock

**Future lockset** of a lock: the set of locks that will be obtained before this lock is released
Deadlock Avoidance Idea on an Example

\{ \text{lock}(x); \ldots \text{lock}(y); \ldots \} \parallel \{ \text{lock}(y); \ldots \text{lock}(x); \ldots \}
Deadlock Avoidance Idea on an Example

\[
\{ \text{lock}_y(x); \ldots \text{lock}_{\emptyset}(y); \ldots \} \parallel \{ \text{lock}_x(y); \ldots \text{lock}_{\emptyset}(x); \ldots \}
\]

only \( y \) is locked here

only \( x \) is locked here

At run-time, the lock annotation is checked

\[\begin{align*}
\text{thread 1} & \text{ tries to lock } x, \text{ with future lockset } \{y\} \\
\text{success!} \\
\text{thread 2} & \text{ tries to lock } y, \text{ with future lockset } \{x\} \\
\text{block!} \\
\text{Lock } y \text{ is available, but lock } x \text{ is held by thread 1} \\
\text{granting } y \text{ to thread 2 may lead to a deadlock!}
\end{align*}\]
Deadlock Avoidance Idea on an Example

\[
\{ \text{lock}_y(x); \ldots \text{lock}_\emptyset(y); \ldots \} \parallel \{ \text{lock}_x(y); \ldots \text{lock}_\emptyset(x); \ldots \}
\]

only \( y \) is locked here

only \( x \) is locked here

At run-time, the lock annotation is checked
Deadlock Avoidance Idea on an Example

At run-time, the lock annotation is checked

- thread 1 tries to lock \( x \), with future lockset \( \{ y \} \)
Deadlock Avoidance Idea on an Example

\[
\{ \text{lock}_y(x); \ldots \text{lock}_\emptyset(y); \ldots \} \parallel \{ \text{lock}_x(y); \ldots \text{lock}_\emptyset(x); \ldots \} \\
\text{only } y \text{ is locked here} \quad \text{only } x \text{ is locked here}
\]

At run-time, the lock annotation is checked

- thread 1 tries to lock \( x \), with future lockset \( \{y\} \) \hspace{1cm} \text{success!}
Deadlock Avoidance Idea on an Example

\[
\{ \text{lock}_{y}(x); \ldots \text{lock}_{\emptyset}(y); \ldots \} \ || \ \{ \text{lock}_{x}(y); \ldots \text{lock}_{\emptyset}(x); \ldots \} \\
\text{only } y \text{ is locked here} \hspace{1cm} \text{only } x \text{ is locked here}
\]

At run-time, the lock annotation is checked

- thread 1 tries to lock \( x \), with future lockset \( \{y\} \) \hspace{1cm} success!
- thread 2 tries to lock \( y \), with future lockset \( \{x\} \)
Deadlock Avoidance Idea on an Example

\[
\begin{align*}
\{ \text{lock}_y(x); \ldots \text{lock}_\emptyset(y); \ldots \} & \quad \| \quad \{ \text{lock}_x(y); \ldots \text{lock}_\emptyset(x); \ldots \} \\
\text{only } y \text{ is locked here} & \quad \| \quad \text{only } x \text{ is locked here}
\end{align*}
\]

At run-time, 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!
Deadlock Avoidance Idea on an Example

\[
\begin{align*}
\{ \text{lock}_y(x); & \ldots \text{lock}_\emptyset(y); \ldots \} \parallel \{ \text{lock}_x(y); & \ldots \text{lock}_\emptyset(x); \ldots \} \\
\text{only } y \text{ is locked here} & \quad \text{only } x \text{ is locked here}
\end{align*}
\]

At run-time, 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!
Code from Linux’s EFS

```c
linux/fs/efs/namei.c:

59   efs_lookup(struct inode *dir, struct dentry *dentry) {
60       efs_ino_t inodenum;
61       struct inode * inode = NULL;
62
63       lock_kernel();
64       inodenum = efs_find_entry(dir, dentry->d_name.name, 
                                dentry->d_name.len);
65       if (inodenum) {
66           if (!(inode = iget(dir->i_sb, inodenum))) {
67               unlock_kernel();
68               return ERR_PTR(-EACCES);
69           }
70       }
71       unlock_kernel();
72
73       d_add(dentry, inode);
74       return NULL;
75     }
```
More Code from Linux

```
linux-2.6-kdbg.git/fs/udf/dir.c:

static int udf_readdir(struct file *filp, ..., filldir_t filldir) {
  struct inode *dir = filp->f_path.dentry->d_inode;
  int result;

  lock_kernel();

  if (filp->f_pos == 0) {
    if (filldir(dirent, ".", 1, ..., dir->i_ino, DT_DIR) < 0) {
      unlock_kernel();
      return 0;
    }
    filp->f_pos++;
  }

  result = do_udf_readdir(dir, filp, filldir, dirent);
  unlock_kernel();
  return result;
}
```
**Locking Patterns**

**Block**

```
foo(a, b) {
    lock(a);
    lock(b);
    ...
    unlock(b);
    unlock(a);
}
```
## Locking Patterns

### Block Structured

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

### Stack Based Same Function

```c
foo(a) {
    lock(a);
    if (...) {
        lock(b);
        unlock(b);
        unlock(a);
        return;
    }
    ...
    unlock(a);
    return;
}
```
### Locking Patterns

**Block Structured**

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

**Stack Based Same Function**

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

**Stack Based Diff Function**

```c
bar(x) {
  lock(x);
}
foo(a) {
  bar(a);
  if (...) {
    unlock(a);
    return;
  }
  ...
  unlock(a);
  return;
}
```
Locking Patterns

Block

foo(a, b) {
  lock(a);
  lock(b);
  ...
  unlock(b);
  unlock(a);
}

Structured

Stack Based

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

Same Function

Stack Based

bar(x) {
  lock(x);
}

Diff Function

foo(a) {
  bar(a);
  if (...) {
    unlock(a);
    return;
  }
  ...
  unlock(a);
  return;
}

Unstructured

foo(a, b) {
  lock(a);
  lock(b);
  ...
  unlock(a);
  unlock(b);
}

P. Gerakios, N. Papaspyrou, K. Sagonas, P. Vekris

Dynamic Deadlock Avoidance in Low-level Languages
Using a **big codebase** (~ 100 big projects using C/pthreads), we gathered statistics on locking patterns

<table>
<thead>
<tr>
<th>Locking Pattern</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block Structured</td>
<td>36.67%</td>
</tr>
<tr>
<td>Stack-Based (same function)</td>
<td>32.22%</td>
</tr>
<tr>
<td>Stack-Based (diff function)</td>
<td>20.00%</td>
</tr>
<tr>
<td>Unstructured</td>
<td>11.11%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>100.00%</strong></td>
</tr>
</tbody>
</table>
Our Approach

To support unstructured locking, we have to

- track the order of `lock` and `unlock` operations
- annotate `lock` operations with a “continuation effect”

```plaintext
foo(x, y, z) { lock[y+, x-, z+, z-, y-](x); x := x + 42;
lock[x-, z+, z-, y-](y); y := y + x;
unlock(x);
lock[z-, y-](z); z := z + y;
unlock(z);
unlock(y);
... }

bar() { ... foo(a, a, b); ... }
```
Our Approach

To support unstructured locking, we have to

- track the order of `lock` and `unlock` operations
- annotate `lock` operations with a “continuation effect”

\[
\begin{align*}
\text{lock}_{[a+,a-,b+,b-,a-]}(a); & \quad a := a + 42; \\
\text{lock}_{[a-,b+,b-,a-]}(a); & \quad a := a + a; \\
\text{unlock}(a); & \\
\text{lock}_{[b-,a-]}(b); & \quad b := b + a; \\
\text{unlock}(b); & \\
\text{unlock}(a)
\end{align*}
\]

After substitution, the continuation effects are still valid! Future locksets are then correctly calculated.
Lockset Calculation

Compute **future lockset** at **run-time** using **lock annotations**

Input: $a+$ \(\rightarrow\) $a+, a-, b+, b-, a-, \ldots$

- lock operation
- continuation effect

\[
\text{lockset} = \{a, b\}
\]

but effects must not be intra-procedural!

what happens if the matching unlock operation occurs after the function returns?
**Lockset Calculation**

Compute **future lockset** at run-time using *lock* annotations

**Input:**

- $a+$
- $a+, a-, b+, b-, a-, \ldots$

- **lock operation**
- **continuation effect**

- Start with an empty future lockset
Lockset Calculation

Compute future lockset at run-time using lock annotations

Input: \( a+ \) \( a+, a-, b+, b-, a-, \ldots \)

- 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 run-time using lock annotations

Input: \( a^+ \) \( a^+, a^-, b^+, b^-, a^-, \ldots \)

- lock operation
- 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 run-time using *lock* annotations

Input: \( a+ \) \( a+, a-, b+, b-, a-, . . . \)

- lock operation
- 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

\[
\text{lockset} = \{ \}
\]
Lockset Calculation

Compute future lockset at run-time using lock annotations

Input: \( a^+ \), \( a^-, b^+, b^-, a^- \), ...

- 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 run-time using lock annotations**

**Input:**

- $a+$
- $a-, b+, b-, a-$
- ...  

- lock operation
- 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 run-time using **lock annotations**

**Input:**

- $a+$ (lock operation)
- $a+, a-, b+, b-, a-, ...$ (continuation effect)

1. Start with an empty future lockset.
2. Traverse the continuation effect until the matching unlock operation (while there are more $a+$ than $a-$).
3. Add the locations being locked to the future lockset.

$\text{lockset} = \{ a, b \}$
**Lockset Calculation**

Compute **future lockset** at run-time using **lock annotations**

**Input:**

- $a+$
- $a+, a-, b+, b-, a-$, ...

- **lock operation**
- **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

\[
\text{lockset} = \{ a, b \}
\]
### Lockset Calculation

Compute **future lockset** at run-time using *lock* annotations

**Input:**

\[ a^+, a^-, b^+, b^-, a^-, \ldots \]

- lock operation
- 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

\[
\text{lockset} = \{ \; a, \; b \; \}
\]
**Lockset Calculation**

Compute future lockset at run-time using *lock* annotations

**Input:**
- $a+$
- $a+, a-, b+, b-, a-, \ldots$

- 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 run-time 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 run-time 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 run-time stack
- Lockset calculation may examine the stack

```c
void f() { g()[z+];
    lock[](z); }

void g() { lock[y+,y-](x);
    lock[y-](y);
    unlock(y); }

m() { f()[z-,x-]; unlock(z); unlock(x); }```

P. Gerakios, N. Papaspyrou, K. Sagonas, P. Vekris Dynamic Deadlock Avoidance in Low-level Languages 15 / 26
Inter-procedural Effects

- Function applications are also annotated with a “continuation effect”
- When a function is applied, the continuation effect is pushed on a run-time stack
- Lockset calculation may examine the stack

```c
void f() { g()[z+];
    lock[](z); }

void g() { lock[y+,y-](x);
    lock[y-](y);
    unlock(y); }

m() { f()[z-,x-]; unlock(z); unlock(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 run-time stack
- Lockset calculation may examine the stack

```c
void f() { g()[z+];
   lock[](z); }

void g() { lock[y+,y-](x);
   lock[y-](y);
   unlock(y); }

m() { f()[z-,x-]; unlock(z); unlock(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 run-time stack
- Lockset calculation may examine the stack

```c
void f() {
    g()[z+];
    lock[](z);
}

void g() {
    lock[y+, y-](x);
    lock[y-](y);
    unlock(y);
}

m() { f()[z-, x-]; unlock(z); unlock(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 run-time stack
- Lockset calculation may examine the stack

```c
void f() {
    g()[z+];
    lock[](z);
}

void g() {
    lock[y+,y-](x);
    lock[y-](y);
    unlock(y);
}

m() {
    f()[z-,x-]; unlock(z); unlock(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 run-time stack
- Lockset calculation may examine the stack

```c
void f() { g()[z+];
    lock[](z); }

void g() { lock[y+,y-](x);
    lock[y-](y);
    unlock(y); }

m() { f()[z-,x-]; unlock(z); unlock(x); }
```

Stack

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>z+</td>
</tr>
<tr>
<td>z-</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>y+</td>
</tr>
<tr>
<td>y-</td>
</tr>
</tbody>
</table>

P. Gerakios, N. Papaspyrou, K. Sagonas, P. Vekris Dynamic Deadlock Avoidance in Low-level Languages 15 / 26
Inter-procedural Effects

- Function applications are also annotated with a “continuation effect”
- When a function is applied, the continuation effect is pushed on a run-time stack
- Lockset calculation may examine the stack

```c
void f() { g()[z+];
    lock[](z); }

void g() { lock[y+,y-](x);
    lock[y-](y);
    unlock(y); }

m() { f()[z-,x-]; unlock(z); unlock(x); }
```

Stack

```
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>z+</td>
</tr>
<tr>
<td>z-, x-</td>
</tr>
</tbody>
</table>
```

Lock/Continuation

```
x+  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 run-time stack
- Lockset calculation may examine the stack

```c
void f() { g()[z+];
    lock[](z); }

void g() { lock[y+,y-](x);
    lock[y-](y);
    unlock(y); }

m() { f()[z-,x-]; unlock(z); unlock(x); }

Stack
    z+
    z-, x-

Lock/Continuation
    x+ y+, y-

lockset = { y }```

P. Gerakios, N. Papaspyrou, K. Sagonas, P. Vekris
Dynamic Deadlock Avoidance in Low-level Languages
Inter-procedural Effects

- Function applications are also annotated with a “continuation effect”
- When a function is applied, the continuation effect is pushed on a run-time stack
- Lockset calculation may examine the stack

```c
void f() { g()[z+];
    lock[](z); }

void g() { lock[y+,y-](x);
    lock[y-](y);
    unlock(y); }

m() { f()[z-,x-]; unlock(z); unlock(x); }
```

Stack

- `z+`
- `z-, x-`

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 run-time stack.
- Lockset calculation may examine the stack.

```c
void f() { g()[z+];
    lock[](z); }

void g() { lock[y+,y-](x);
    lock[y-](y);
    unlock(y); }

m() { f()[z-,x-]; unlock(z); unlock(x); }
```

Stack

<table>
<thead>
<tr>
<th>+</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>z</td>
<td>x</td>
</tr>
</tbody>
</table>

Lock/Continuation

| x+ | 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 run-time stack
- Lockset calculation may examine the stack

```c
void f() { g()[z+];
    lock[](z); }

void g() { lock[y+,y-](x);
    lock[y-](y);
    unlock(y); }

m() { f()[z-,x-]; unlock(z); unlock(x); }
```

Stack

- z+
- z-, x-

Lock/Continuation

- x+ 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 run-time stack
- Lockset calculation may examine the stack

```c
void f() { g()[z+];
    lock[](z); }

void g() { lock[y+,y-](x);
    lock[y-](y);
    unlock(y); }

m() { f()[z-,x-]; unlock(z); unlock(x); }
```

Stack

- Lock/Continuation
  - `x+`  
  - `y+`, `y-`

Lockset = { y, z }
Conditional Expressions

\[ \text{if} \ (e) \ \text{then} \ e_1 \ \text{else} \ e_2 \]

- How can we type-check conditionals?
Conditional Expressions

if \( (e) \) then \( e_1 \) else \( e_2 \)

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

```plaintext
lock(x);
if (condition) {
    lock(y); ...; unlock(y);  
    effect: \( y+ \), \( y- \)
}
unlock(x);
```

conservative, require:

```
effect(e_1) = effect(e_2)
```

we require:

```
overall(effect(e_1)) = overall(effect(e_2))
```

see TLDI'11 paper for treatment of loops/recursion
Conditional Expressions

if \( e \) then \( e_1 \) else \( e_2 \)

- How can we type-check conditionals?
- Consider:
  
  ```
  lock(x);
  if (condition) {
    lock(y); ...; unlock(y);  
    effect: y+, y-
  }
  unlock(x);
  ```

- Conservative, require: \( \text{effect}(e_1) = \text{effect}(e_2) \)
Conditional Expressions

if \( (e) \) then \( e_1 \) else \( e_2 \)

- How can we type-check conditionals?
- Consider:
  
  ```
  lock(x);
  if (condition) {
    lock(y); ...; unlock(y);  
    effect: \( y +, y- \)
  }                    
  effect: empty  
  unlock(x);
  ```

- Conservative, require: \( \text{effect}(e_1) = \text{effect}(e_2) \)
- We require: \( \text{overall}(\text{effect}(e_1)) = \text{overall}(\text{effect}(e_2)) \)
Conditional Expressions

```
if (e) then e₁ else e₂
```

- How can we type-check conditionals?
- Consider:
  ```
  lock(x);
  if (condition) {
    lock(y); ...; unlock(y);       effect: y+, y−
  }
  unlock(x);
  ```

- Conservative, require:  
  \[ \text{effect}(e₁) = \text{effect}(e₂) \]

- We require:  
  \[ \text{overall(effect}(e₁)) = \text{overall(effect}(e₂)) \]

- See TLDI’11 paper for treatment of loops/recursion
A Tool for C/pthreads

- Input: C program annotation free

- At compile time, perform a field-sensitive, context-sensitive pointer analysis
- Infer annotations/effects
- Instrument code with continuation effects
- Link program with a run-time system
- Overrides pthread library
- Utilizes the effects in the code to compute future locksets
- Grant locks in a way that avoids deadlocks
A Tool for C/pthreads

- Input: C program annotation free
- At compile time
  - perform a field-sensitive, context-sensitive pointer analysis
  - infer annotations/effects
A Tool for C/pthreads

- Input: C program annotation free

- At compile time
  - perform a field-sensitive, context-sensitive pointer analysis
  - infer annotations/effects
  - instrument code with continuation effects

- Link program with a run-time system
  - overrides pthread library
  - utilizes the effects in the code to
    - compute future locksets
    - grant locks in a way that avoids deadlocks
A Tool for C/pthreads

- **Input:** C program *annotation free*

- **At compile time**
  - perform a field-sensitive, context-sensitive pointer analysis
  - infer annotations/effects
  - instrument code with continuation effects

- **Link** program with a run-time system
  - overrides pthread library
A Tool for C/pthreads

- **Input**: C program **annotation free**
- **At compile time**
  - perform a field-sensitive, context-sensitive pointer analysis
  - infer annotations/effects
  - instrument code with continuation effects
- **Link** program with a run-time system
  - overrides pthread library
  - utilizes the effects in the code to
    - compute future locksets
    - grant locks in a way that avoids deadlocks
Static Analysis: Inference

- **Call-graph**: bottom-up traversal
Static Analysis: Inference

- **Call-graph**: bottom-up traversal
- **Loops**:
  - may have any number of lock/unlock operations
  - lock counts upon loop exit must equal counts before the loop entry

Indirect calls: effect \( ((\ast f)(x)) \):

- pointer analysis

\[ f \mapsto \{ c_1, \ldots, c_n \} \]

\[ \text{effect}((\ast f)(x)) = \text{effect}(c_1(x)) \ldots \text{effect}(c_n(x)) \]
Static Analysis: Inference

- **Call-graph:** bottom-up traversal
- **Loops:**
  - may have any number of lock/unlock operations
  - lock counts upon loop exit must equal counts before the loop entry
- **Indirect calls:** \( \text{effect}(\ast f(x)) \):
Static Analysis: Inference

- **Call-graph**: bottom-up traversal
- **Loops**:
  - may have any number of lock/unlock operations
  - lock counts upon loop exit must equal counts before the loop entry
- **Indirect calls**: \( \text{effect}((\ast f)(x)) \):
  - pointer analysis \( f \mapsto \{ c_1, \ldots, c_n \} \)
Static Analysis: Inference

- **Call-graph**: bottom-up traversal
- **Loops**: 
  - may have any number of lock/unlock operations 
  - lock counts upon loop exit must equal counts before the loop entry
- **Indirect calls**: \( \text{effect}(\ast f(x)):\) 
  - pointer analysis \( f \mapsto \{ c_1, \ldots, c_n \} \) 
  - \( \text{effect}(\ast f(x)) = \text{effect}(c_1(x)) \ldots \text{effect}(c_n(x)) \)
- **Pointer analysis for lock handle pointers**
Static Analysis: Status and Limitations

Support for:

- pointers to global lock handles
- dynamically allocated lock handles (heap + stack)

Requires no programmer-supplied annotations of any sort
Static Analysis: Status and Limitations

Support for:

- pointers to global lock handles
- dynamically allocated lock handles (heap + stack)

Requires no programmer-supplied annotations of any sort

No support for:

- non C code
- non-local jumps
- pointer arithmetic on pointers containing or pointing to locks
**Locking Algorithm**

Upon a $\text{lock}(x)$ with future lockset $L$:

1. Check whether all locks in $L$ are available
2. If not, wait
3. Otherwise, *tentatively acquire* lock $x$
4. Check again $L$: if any lock in $L$ is unavailable
   - release $x$
   - wait on that unavailable lock
### Evaluation: On bigger C programs

<table>
<thead>
<tr>
<th>benchmark</th>
<th>run in</th>
<th>user</th>
<th>system</th>
<th>elapsed</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>curlftpfs</td>
<td>C</td>
<td>0.002</td>
<td>0.758</td>
<td>33.450</td>
<td>0.982</td>
</tr>
<tr>
<td></td>
<td>C+da</td>
<td>0.000</td>
<td>0.680</td>
<td>32.862</td>
<td></td>
</tr>
<tr>
<td>flam3</td>
<td>C</td>
<td>63.660</td>
<td>3.910</td>
<td>49.050</td>
<td>1.003</td>
</tr>
<tr>
<td></td>
<td>C+da</td>
<td>67.860</td>
<td>3.640</td>
<td>49.200</td>
<td></td>
</tr>
<tr>
<td>migrate-n</td>
<td>C</td>
<td>5545.311</td>
<td>4631.341</td>
<td>4138.070</td>
<td>1.118</td>
</tr>
<tr>
<td></td>
<td>C+da</td>
<td>5334.921</td>
<td>5020.346</td>
<td>4625.670</td>
<td></td>
</tr>
<tr>
<td>ngorca</td>
<td>C</td>
<td>124.846</td>
<td>0.126</td>
<td>8.270</td>
<td>0.996</td>
</tr>
<tr>
<td></td>
<td>C+da</td>
<td>124.467</td>
<td>0.126</td>
<td>8.240</td>
<td></td>
</tr>
<tr>
<td>sshfs-fuse</td>
<td>C</td>
<td>0.000</td>
<td>0.890</td>
<td>20.880</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>C+da</td>
<td>0.000</td>
<td>0.950</td>
<td>20.880</td>
<td></td>
</tr>
<tr>
<td>tgrep</td>
<td>C</td>
<td>13.238</td>
<td>11.639</td>
<td>5.190</td>
<td>1.191</td>
</tr>
<tr>
<td></td>
<td>C+da</td>
<td>14.801</td>
<td>11.655</td>
<td>6.180</td>
<td></td>
</tr>
</tbody>
</table>

Performance of C vs. C+da (C plus deadlock avoidance)
Evaluation: Cosmic Fractal Frames

flan3 workload

Elapsed time in seconds vs. number of collaborating threads.

Original C program
Instrumented C program
Evaluation: File System over SSH
Evaluation: Dining Philosophers

The diagram illustrates the philosophers workload under two different conditions:

- **Original C program** represented by red crosses.
- **Instrumented C program** represented by green squares.

The x-axis represents the number of philosophers, while the y-axis shows the total number of times the philosophers ate, measured in millions.

The graph shows a steady increase in the total number of times the philosophers ate as the number of philosophers increases. The instrumented C program shows a smoother trend compared to the original C program, indicating a potential impact of the instrumentation on the workload.
Concluding Remarks

- A method that guarantees deadlock freedom
  - without imposing a global lock acquisition order
  - unstructured locking primitives

- A tool for C/pthreads
  - completely automatic: no annotations are needed
  - modest run-time overhead for instrumented programs
Thank you!

Questions?