Dynamic Deadlock Avoidance Using Statically Inferred Effects

Kostis Sagonas[1,2]

joint work with

P. Gerakios[1]

N. Papaspyrou[1]

P. Vekris[1,3]

[1] School of ECE, National Technical University of Athens, Greece
[2] Dept. of Information Technology, Uppsala University, Sweden
[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`)

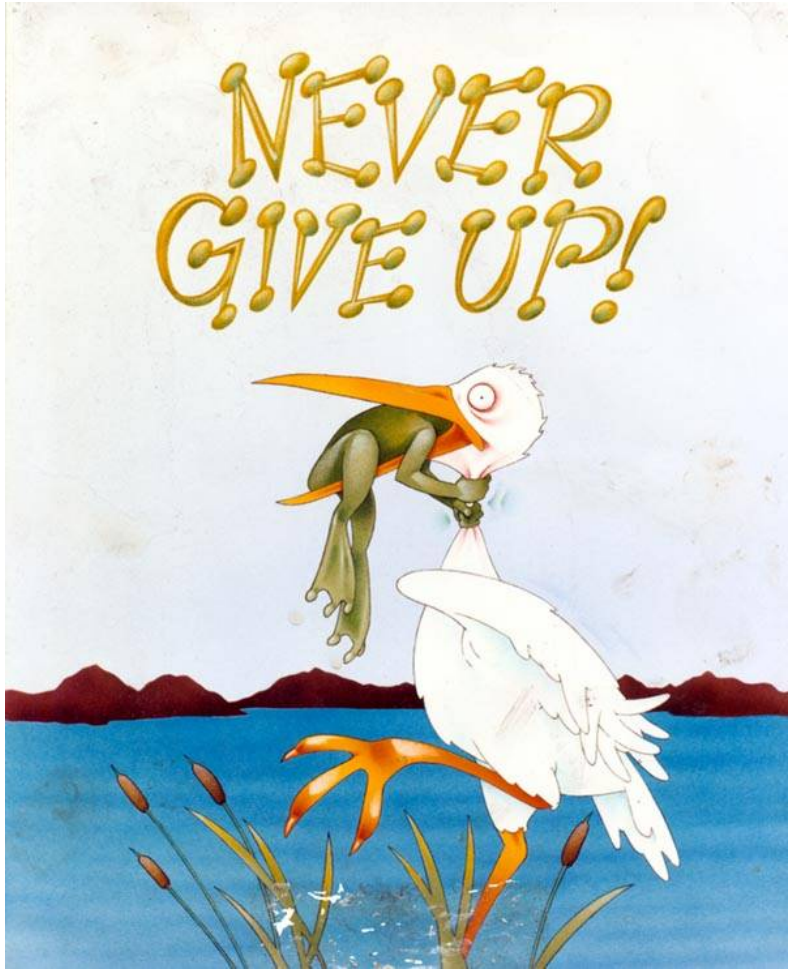
Tool for C/pthreads programs

- 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:

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

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:

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

- no single global order $\Rightarrow$ reject program

Deadlock Avoidance: 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

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

Deadlock Avoidance Idea on an Example

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

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

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

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

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

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

Deadlock Avoidance Idea on an 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 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 $\{x\}$

Deadlock Avoidance Idea on an 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 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

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

At 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

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:

```
188  static int udf_readdir(struct file *filp, ..., filldir_t filldir)
189  {
190    struct inode *dir = filp->f_path.dentry->d_inode;
191    int result;
192
193    lock_kernel();
194
195    if (filp->f_pos == 0) {
196      if (filldir(dirent, ".", 1, ..., dir->i_ino, DT_DIR) < 0) {
197        unlock_kernel();
198        return 0;
199      }
200      filp->f_pos++;
201    }
202
203    result = do_udf_readdir(dir, filp, filldir, dirent);
204    unlock_kernel();
205    return result;
206  }
```

Locking Patterns

Block

Structured

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

Locking Patterns

Block Structured

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

Stack Based Same Function

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

Locking Patterns

Block Structured

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

Stack Based Same Function

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

Stack Based Diff Function

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

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

Locking Patterns

Block Structured

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

Stack Based Same Function

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

Stack Based Diff Function

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

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

Unstructured

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

How C/pthreads Programs use Locks?

Using a big codebase ($\sim$ 100 big projects using C/pthreads), we gathered statistics on locking patterns

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

Our Approach

To support unstructured locking, we have to

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

$$
\begin{array}{ll}
\texttt{foo}(x, y, z)\ \{\ \texttt{lock}_{[y+,x-,z+,z-,y-]}(x); & x := x + 42; \\
\quad \texttt{lock}_{[x-,z+,z-,y-]}(y); & y := y + x; \\
\quad \texttt{unlock}(x); & \\
\quad \texttt{lock}_{[z-,y-]}(z); & z := z + y; \\
\quad \texttt{unlock}(z); & \\
\quad \texttt{unlock}(y); & \\
\quad \ldots\ \}
\end{array}
$$

$$\texttt{bar}()\ \{\ \ldots\ \texttt{foo}(a, a, b);\ \ldots\ \}$$

Our Approach

To support unstructured locking, we have to

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

$$
\begin{array}{ll}
\texttt{lock}_{[a+,a-,b+,b-,a-]}(a); & a := a + 42; \\
\texttt{lock}_{[a-,b+,b-,a-]}(a); & a := a + a; \\
\texttt{unlock}(a); & \\
\texttt{lock}_{[b-,a-]}(b); & b := b + a; \\
\texttt{unlock}(b); & \\
\texttt{unlock}(a) &
\end{array}
$$

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

Lockset Calculation

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

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

- start with an empty future lockset

Lockset Calculation

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

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

- 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 run-time using $lock$ annotations

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

- 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 run-time using $lock$ annotations

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

- 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

$$\text{lockset} = \{\quad\}$$

Lockset Calculation

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

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

- 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 run-time using $lock$ annotations

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

- 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 run-time using *lock* annotations

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

- 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

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

Lockset Calculation

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

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

- 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

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

Lockset Calculation

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

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

- 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

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

Lockset Calculation

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

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

- 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 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

$$\texttt{void}\ f()\ \{\ g()_{[z+]};\ \texttt{lock}_{[]}(z);\ \}$$

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

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

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

```
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

```
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

$z-,\ x-$

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

```
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

$z-,\ x-$

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

```
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

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

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

```
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

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

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

```
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 = { }

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

```
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

```
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

```
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

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

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 run-time stack
- Lockset calculation may examine the stack

```
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

```
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 }

Conditional Expressions

`if` (e) `then` e_1 `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:

```
lock(x);
if (condition) {
   lock(y); ...; unlock(y);        effect: y+, y−
}                                  effect: empty
unlock(x);
```

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: $\texttt{effect}(e_1) = \texttt{effect}(e_2)$

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: $\texttt{effect}(e_1) = \texttt{effect}(e_2)$
- We require: $\texttt{overall}(\texttt{effect}(e_1)) = \texttt{overall}(\texttt{effect}(e_2))$

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: `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

A Tool for C/pthreads

- Input: C program annotation free

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

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

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`$((*f)(x))$:

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: `effect`$((*f)(x))$:
  - 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: $\texttt{effect}((*f)(x))$:
  - pointer analysis $f \mapsto \{c_1, \ldots, c_n\}$
  - $\texttt{effect}((*f)(x)) = \texttt{effect}(c_1(x))\ ?\ \ldots\ ?\ \texttt{effect}(c_n(x))$
- 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 `lock`(x) with future lockset L:

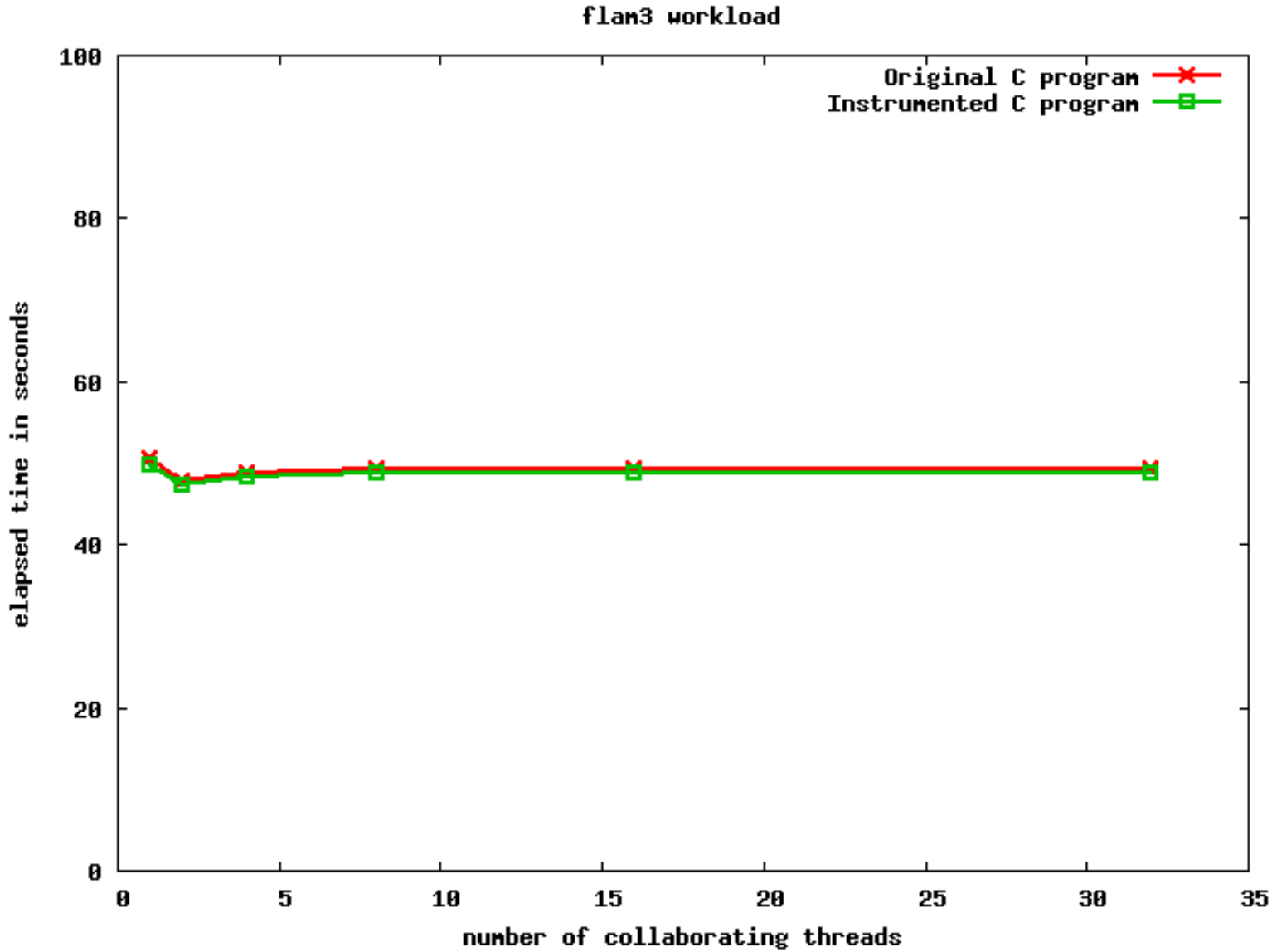
1. Check whether all locks in L are available
2. If not, wait
3. Otherwise, *tentatively acquire* lock x
4. Check again L: if any lock in L is unavailable
   - release x
   - wait on that unavailable lock

Evaluation: On bigger C programs

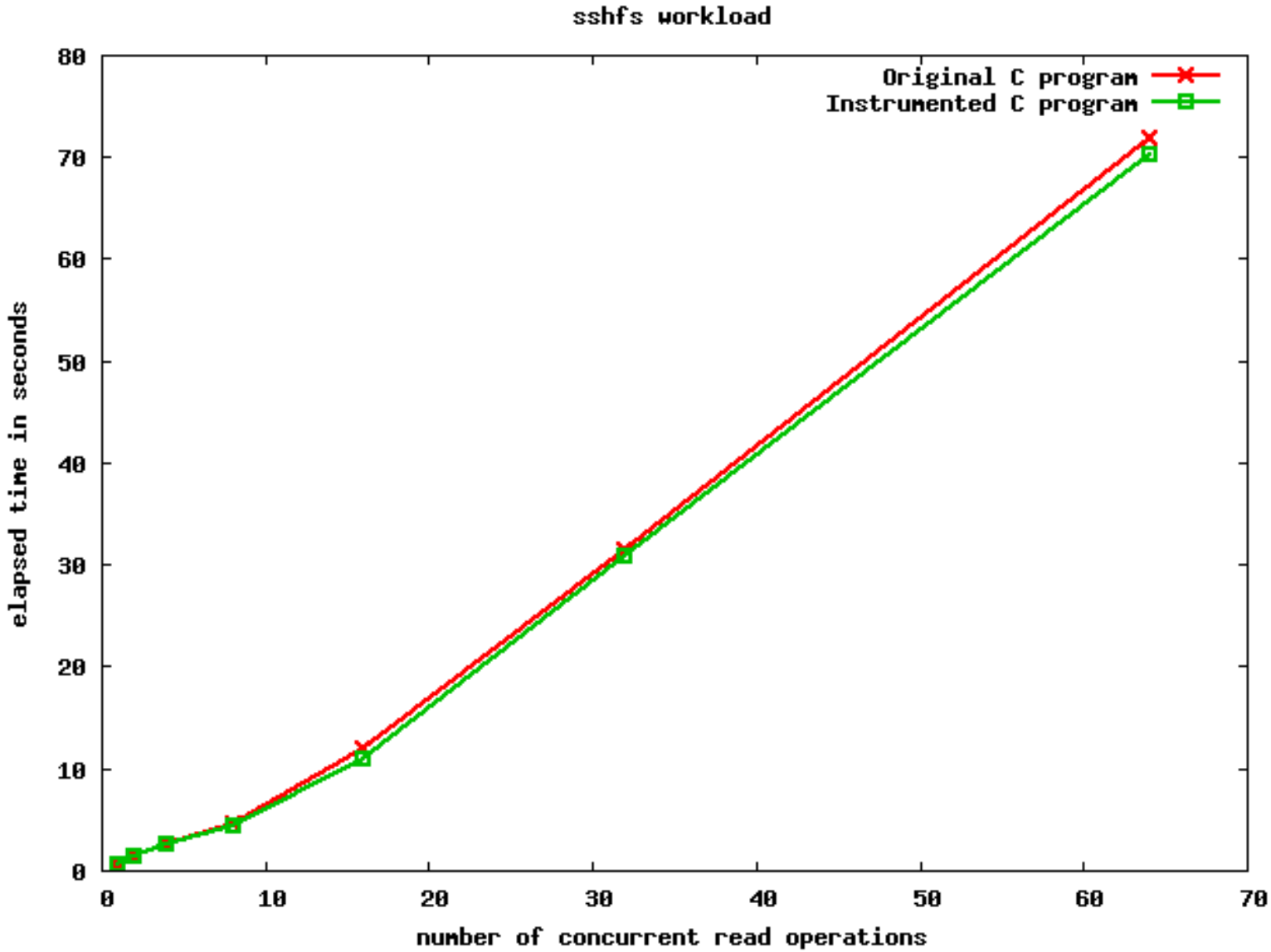
benchmark	run in	user	system	elapsed	ratio
curlftpfs	C	0.002	0.758	33.450	**0.982**
	C+da	0.000	0.680	32.862	
flam3	C	63.660	3.910	49.050	**1.003**
	C+da	67.860	3.640	49.200	
migrate-n	C	5545.311	4631.341	4138.070	**1.118**
	C+da	5334.921	5020.346	4625.670	
ngorca	C	124.846	0.126	8.270	**0.996**
	C+da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

Performance of C vs. C+da (C plus deadlock avoidance)

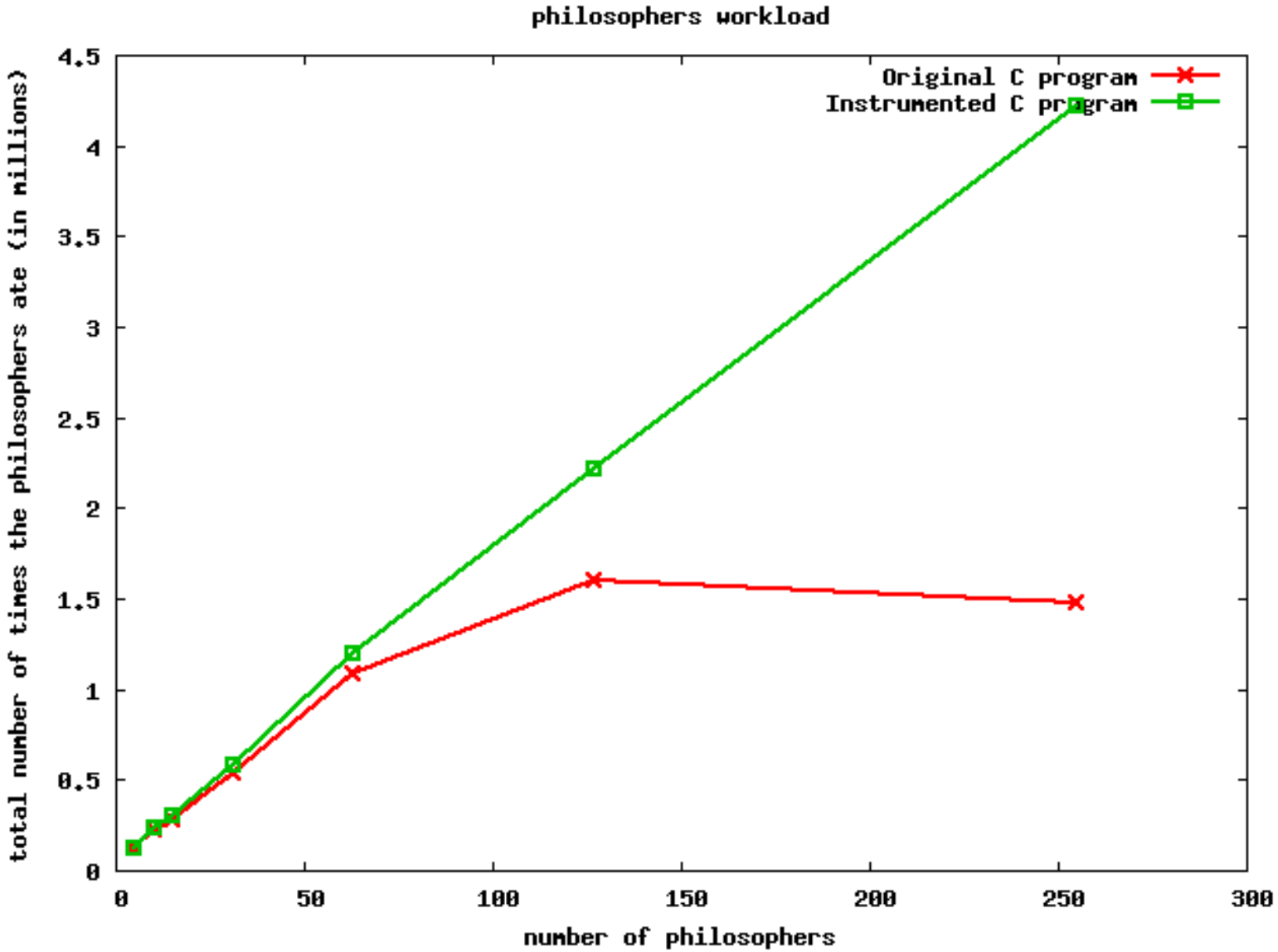
Evaluation: Cosmic Fractal Frames



Evaluation: File System over SSH



Evaluation: Dining Philosophers



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?