LOCKPICK: Lock Inference for Atomic Sections

Jeffrey S. Foster
Michael Hicks
Polyvios Pratikakis

University of Maryland, College Park
Concurrent programming is “notoriously difficult”

More parallelism is good, too much is wrong

Less parallelism is easier, but it slows down the program

Synchronization is done using locks

Locks are difficult to program

Alternative, higher level synchronization abstraction: atomic sections
int x, y;

thread1() {
    atomic {
        x = 42;
        y = 43;
    }
}

thread2() {
    atomic {
        x = 44;
    }
}

Atomic sections usually use optimistic concurrency

This work: atomic sections with pessimistic concurrency
LOCKPICK at a glance

- Create a mutex $\ell_\rho$ for each memory location $\rho$
- Create a total ordering on all $\ell_\rho$ to avoid deadlock
- For every atomic block, if $\rho$ is referenced, then acquire $\ell_\rho$ at the beginning
- Maintain maximum parallelism (for the given points-to analysis)
LOCKPICK at a glance

- Create a mutex $\ell_p$ for each memory location $p$
- Create a total ordering on all $\ell_p$ to avoid deadlock
- For every atomic block, if $p$ is referenced, then acquire $\ell_p$ at the beginning

- Maintain maximum parallelism (for the given points-to analysis)

**Inefficient:** large number of locations $\Rightarrow$ large number of locks
LOCKPICK at a glance

- Find all memory locations $\rho$ that are shared between threads
- Create a mutex $\ell_\rho$ for each memory location $\rho$
- Create a total ordering on all $\ell_\rho$ to avoid deadlock
- For every atomic block, if $\rho$ is referenced, then acquire $\ell_\rho$ at the beginning

- Maintain maximum parallelism (for the given points-to analysis)
LOCKPICK at a glance

- Find all memory locations \( \rho \) that are shared between threads
- Create a mutex \( \ell_\rho \) for each memory location \( \rho \)
- Create a total ordering on all \( \ell_\rho \) to avoid deadlock
- For every atomic block, if \( \rho \) is referenced, then acquire \( \ell_\rho \) at the beginning

- Maintain maximum parallelism (for the given points-to analysis)

Inefficient: many locations are always referenced together
Find all memory locations $\rho$ that are shared between threads

Create a mutex $\ell_\rho$ for each memory location $\rho$

Create a total ordering on all $\ell_\rho$ to avoid deadlock

For every atomic block, if $\rho$ is referenced, then acquire $\ell_\rho$ at the beginning

Find and remove unnecessary locks

Maintain maximum parallelism (for the given points-to analysis)
Example

```c
int x, y;

thread1() { atomic {
    x = 42;
    y = 43;
}

thread2() { atomic {
    x = 44;
}
```

Whenever \( L_y \) is locked, \( L_x \) is also locked. \( L_x \) dominates \( L_y \); only \( L_y \) is unnecessary, only adds overhead. Optimization: when \( r \) dominates \( r_0 \), protect \( r_0 \) with `\( r \)`.

Lock Inference for Atomic Sections – p. 5/11
Example

```c
int x, y;
mutex_t Lx, Ly;

thread1() { atomic {
    x = 42;
    y = 43;
} }

thread2() { atomic {
    x = 44;
} }
```

Whenever `Ly` is locked, `Lx` is also locked. `Lx` dominates `Ly` is unnecessary, only adds overhead. Optimization: when `r` dominates `r`, protect `r` with `r`.

Lock Inference for Atomic Sections – p. 5/11
Example

int x, y;
mutex_t Lx, Ly;
thread1() { atomic {
    lock(Lx); lock(Ly);
    x = 42;
    y = 43;
}
}

thread2() { atomic {
    x = 44;
}
}
Example

```c
int x, y;
mutex_t Lx, Ly;

thread1() { atomic {
    lock(Lx); lock(Ly);
    x = 42;
    y = 43;
    unlock(Lx); unlock(Ly);
} }

thread2() { atomic {
    x = 44;
} }
```

Whenever `Ly` is locked, `Lx` is also locked. `Lx` dominates `Ly`, so `Ly` is unnecessary, only adds overhead. Optimization: when `r` dominates `r`, protect `r` with `r`. Lock Inference for Atomic Sections – p. 5/11
Example

```c
int x, y;
mutex_t Lx, Ly;

thread1() { atomic {
    lock(Lx); lock(Ly);
    x = 42;
    y = 43;
    unlock(Lx); unlock(Ly);
}
}

thread2() { atomic {
    lock(Lx);
    x = 44;
    unlock(Lx);
}
}
```

Whenever Ly is locked, Lx is also locked. Lx dominates Ly, so Ly is unnecessary, only adds overhead.

Optimization: when r dominates r0, protect r0 with `r`.
Example

```c
int x, y;
mutex_t Lx, Ly;

thread1() { atomic {
    lock(Lx); lock(Ly);
    x = 42;
    y = 43;
    unlock(Lx); unlock(Ly);
}
}

thread2() { atomic {
    lock(Lx);
    x = 44;
    unlock(Lx);
}
}
```

- Whenever $L_y$ is locked, $L_x$ is also locked
- $L_x$ dominates $L_y$
- $L_y$ is unnecessary, only adds overhead
- Optimization: when $p$ dominates $p'$, protect $p'$ with $\ell_p$. 
Example: The Dominates Algorithm

```c
int x, y;
thread1() {
    atomic {
        x = 42;
        y = 43;
    }
}

thread2() {
    atomic {
        x = 44;
    }
}
```
int x, y;

thread1() {
    atomic {
        x = 42;
        y = 43;
    }
}

thread2() {
    atomic {
        x = 44;
    }
}

Each atomic section dereferences a set of locations
Example: The Dominates Algorithm

```c
int x, y;
thread1() {
    atomic α₁{
        x = 42;
        y = 43;
    }
}

thread2() {
    atomic {
        x = 44;
    }
}
```

Each atomic section dereferences a set of locations
int x, y;

thread1() {
    atomic \( \alpha_1 \) {
        x = 42;
        y = 43;
    }
}

thread2() {
    atomic \( \alpha_2 \) {
        x = 44;
    }
}

Each atomic section dereferences a set of locations
int x, y;

thread1() {
    atomic α₁{
        x = 42;
        y = 43;
    }
}

thread2() {
    atomic α₂{
        x = 44;
    }
}

Each atomic section dereferences a set of locations. Atomic section α is a set of the locations it dereferences.
Example: The Dominates Algorithm

```c
int x, y;
thread1() {
    atomic α₁{
        x = 42;
        y = 43;
    }
}

thread2() {
    atomic α₂{
        x = 44;
    }
}
```

Each atomic section dereferences a set of locations. Atomic section α is a set of the locations it dereferences: $α₁ = \{x, y\}$, $α₂ = \{x\}$
int x, y;

thread1() {
    atomic α₁{
        x = 42;
        y = 43;
    }
}

thread2() {
    atomic α₂{
        x = 44;
    }
}

Each atomic section dereferences a set of locations. Atomic section α is a set of the locations it dereferences: α₁ = \{x, y\}, α₂ = \{x\}

x > y
Domination algorithm reduces the number of used locks
Always retains maximum parallelism
Sound: it never introduces races
May not find minimum number of locks
Minimizing the number of locks is NP-hard
Proof: reduction from Edge Clique Cover
Example: Limitation of the algorithm

```
atomic {
    x = 1;
y = 2;
}

atomic {
    y = 3;
z = 4;
}

atomic {
    z = 5;
x = 6;
}
```

\[ \alpha_1 = \{x, y\} \quad \alpha_2 = \{y, z\} \quad \alpha_3 = \{x, z\} \]

- No “dominates” relation holds
- No parallelism possible
- The program can be synchronized with one lock
What is shared?

Inefficiency:
- Atomic blocks might dereference many locations
- Only a few are shared between threads

Optimization: Only protect shared locations
- Find continuation effects
- Intersect effects of threads to find shared locations
Continuation Effects: Example

```c
int x, y;
main() {
    x = 1;
    pthread_create(&thread1);
    y = 2;
}
thread1() {
    x = 42;
    y = 43;
}
```
Continuation Effects: Example

```c
int x, y;

main() {
    x = 1;
    pthread_create(&thread1);
    y = 2;
}

thread1() {
    x = 42;
    y = 43;
}
```
Continuation Effects: Example

```c
int x, y;
main() {
    x = 1;
    pthread_create(&thread1);
    y = 2;
}
thread1() {
    x = 42;
    y = 43;
}
```
int x, y;
main() {
    x = 1;
    pthread_create(&thread1);
    y = 2;
}

thread1() {
    x = 42;
    y = 43;
}
int x, y;
main() {
  x = 1;
  pthread_create(&thread1);
  y = 2;
}
thread1() {
  x = 42;
  y = 43;
}
int x, y;
main() {
    x = 1;
    pthread_create(&thread1);
    y = 2;
}
thread1() {
    x = 42;
    y = 43;
}

shared = \varepsilon_4 \cap \varepsilon_6 = \{y\}
Conclusions

Contributions:

- Atomic sections can be implemented with pessimistic concurrency
- Heuristic algorithm to reduce number of locks without losing parallelism
- Finding the minimum number of locks is NP-hard
- Precise sharing analysis to further reduce needed locks

- Implementation under construction: LOCKPICK
- Fine grain locking for shared data-structures