LOCKPICK: Lock Inference for Atomic Sections

Jeffrey S. Foster
Michael Hicks
Polyvios Pratikakis

University of Maryland, College Park

Introduction

- 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

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 ℓ_ρ for each memory location ρ
- Create a total ordering on all ℓ_ρ to avoid deadlock
- For every atomic block, if ρ is referenced, then acuire ℓ_ρ at the beginning

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

LOCKPICK at a glance

- Create a mutex ℓ_ρ for each memory location ρ
- Create a total ordering on all ℓ_ρ to avoid deadlock
- For every atomic block, if ρ is referenced, then acuire ℓ_ρ 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 ρ that are shared between threads
- Create a mutex ℓ_ρ for each memory location ρ
- Create a total ordering on all ℓ_ρ to avoid deadlock
- For every atomic block, if ρ is referenced, then acuire ℓ_ρ at the beginning

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

LOCKPICK at a glance

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

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

Inefficient: many locations are always referenced together

LOCKPICK at a glance

- Find all memory locations ρ that are shared between threads
- Create a mutex ℓ_ρ for each memory location ρ
- Create a total ordering on all ℓ_ρ to avoid deadlock
- For every atomic block, if ρ is referenced, then acuire ℓ_ρ at the beginning
- Find and remove unnecessary locks
- Maintain maximum parallelism (for the given points-to analysis)

Example

```
int x, y;

thread1() { atomic {

    x = 42;
    y = 43;

} }

thread2() { atomic {

    x = 44;

} }
```

Example

```
int x, y;
mutex_t Lx, Ly;
 thread1() { atomic {

      x = 42;
      y = 43;

 } }
```

```
thread2() { atomic {

      x = 44;

} }
```

Example

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

 } }
```

```
thread2() { atomic {

     x = 44;

} }
```

Example

```
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;

} }
```

Example

```
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);
} }
```

Example

```
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`
- `Ly` is unnecessary, only adds overhead
- Optimization: when ρ dominates ρ', protect ρ' with ℓ_ρ.

Example: The Dominates Algorithm

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

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

Example: The Dominates Algorithm

```
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

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

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

Each atomic section dereferences a set of locations

Example: The Dominates Algorithm

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

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

Each atomic section dereferences a set of locations

Example: The Dominates Algorithm

```
int x, y;
thread1() {                        thread2() {
   atomic α1{                         atomic α2{
     x = 42;                            x = 44;
     y = 43;                          }
   }                               }
}
```

Each atomic section dereferences a set of locations Atomic section α is a set of the locations it dereferences

Example: The Dominates Algorithm

```
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 $\alpha_1 = \{x, y\}$, $\alpha_2 = \{x\}$

Example: The Dominates Algorithm

```
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 $\alpha_1 = \{x, y\}$, $\alpha_2 = \{x\}$

$x > y$

Remarks

- 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

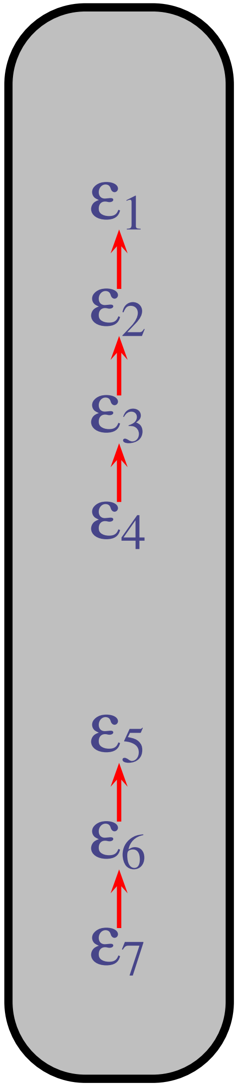
Continuation Effects: Example

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

Continuation Effects: Example

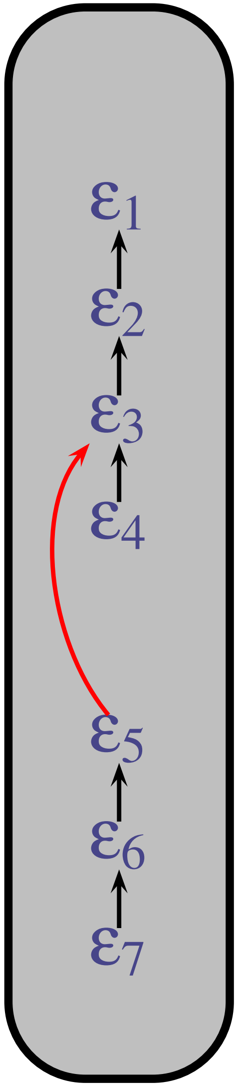
	`int x, y;`
ε_1	`main() {`
ε_2	`    x = 1;`
ε_3	`    pthread_create(&thread1);`
ε_4	`    y = 2;`
	`}`
ε_5	`thread1() {`
ε_6	`    x = 42;`
ε_7	`    y = 43;`
	`}`

Continuation Effects: Example



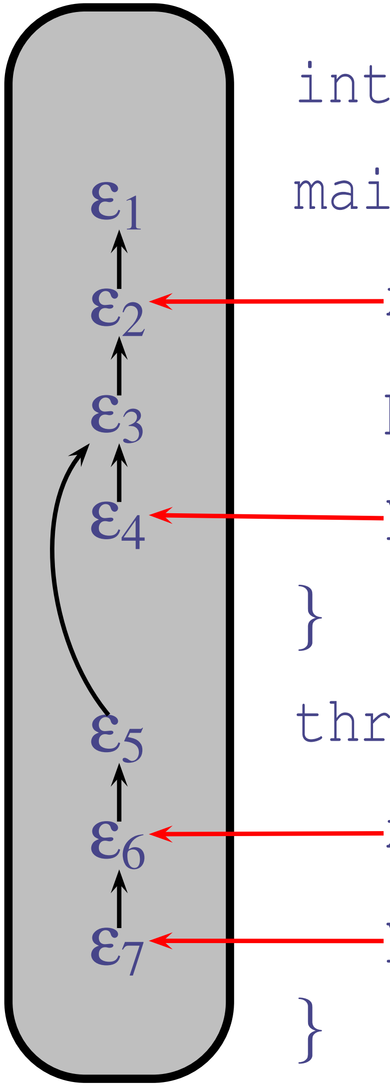

```
int x, y;
main() {
   x = 1;
   pthread_create(&thread1);
   y = 2;
}
thread1() {
   x = 42;
   y = 43;
}
```

Continuation Effects: Example



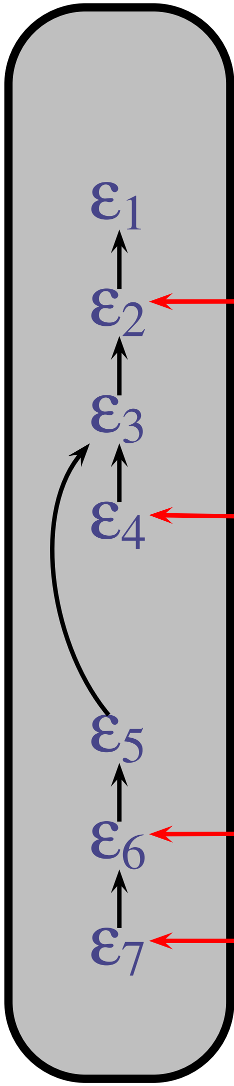

```
int x, y;
main() {
   x = 1;
   pthread_create(&thread1);
   y = 2;
}
thread1() {
   x = 42;
   y = 43;
}
```

Continuation Effects: Example




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

Continuation Effects: Example




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

$$\text{shared} = \varepsilon_4 \cap \varepsilon_6 = \{y\}$$

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