Component-Based Lock Allocation

Richard Halpert    Chris Pickett    Clark Verbrugge

School of Computer Science, McGill University
{rhalpe, cpicke, clump}@sable.mcgill.ca

6th Workshop on Compiler-Driven Performance
CASCON
October 22nd, 2007
- **Critical section**: piece of code that accesses shared state exclusively
- **Lock**: object that guards access to a critical section
- **Lock allocation**: mapping locks to critical sections

Sounds straightforward, but manual approaches are tricky!
class T1 extends Thread
{
    public static Object a;

    run ()
    {
        synchronized (T1.a)
        {
            Main.i++;  // race condition!
        }
    }
}

class T2 extends Thread
{
    public static Object b;

    run ()
    {
        synchronized (T2.b)
        {
            Main.i++;  
        }
    }
}
class T1 extends Thread
{
    public static Object a;
    run()
    {
        synchronized (T1.a)
        {
            synchronized (T2.b)
            {
                Main.i++;
            }
        }
    }
}

class T2 extends Thread
{
    public static Object b;
    run()
    {
        synchronized (T2.b)
        {
            synchronized (T1.a)
            {
                Main.i++;
            }
        }
    }
}
class T1 extends Thread
{
    public static Object a;

    run()
    {
        synchronized (T1.a)
        {
            synchronized (T2.b)
            {
                t1Work();
            }
        }
    }
}

class T2 extends Thread
{
    public static Object b;

    run()
    {
        synchronized (T1.a)
        {
            synchronized (T2.b)
            {
                t2Work();
            }
        }
    }
}
Our approach: *automatic* lock allocation

Goal: simplify concurrent programming

- Remove burden of manual allocation from programmer
- Aim to be *strictly* simpler: no extra language constructs
- Ideal result: automatic allocation performance matches or exceeds manual allocation performance
Our contributions:

- We investigate *component-based* lock allocation:
  - Coarse locking granularity
  - Construct a critical section interference graph
  - One lock per graph component

- Experiment with many static compiler analyses
- Show results for small and large Java benchmarks

The technique often performs well:

- Matches manual allocation performance on 2, 4, 8-way hardware for mtrt (SPEC JVM98), lusearch and xalan (DaCapo), and SPEC JBB2005.
Outline

1. Introduction
2. Design
3. Experimental Results
4. Conclusions and Future Work
class G {
    public static int X, Y;
}

class T1 extends Runnable {
    run() {
        synchronized(...) { // CS1
            G.Y = G.X;
        }
        synchronized(...) { // CS2
            G.X = G.X + 1;
        }
    }
}

class T2 extends Runnable {
    run() {
        synchronized(...) { // CS3
            int a = G.Y;
        }
    }
}
class G {
    public static int X, Y;
}

class T1 extends Runnable {
    run() {
        synchronized(...singletonObject...) { // CS1
            G.Y = G.X;
        }
        synchronized(...singletonObject...) { // CS2
            G.X = G.X + 1;
        }
    }
}

class T2 extends Runnable {
    run() {
        synchronized(...singletonObject...) { // CS3
            int a = G.Y;
        }
    }
}
class G {
    public static int X, Y;
}

class T1 extends Runnable {
    run() {
        synchronized(...) { // CS1
            G.Y = G.X; // Read from X, thread-shared
            Write to Y, thread-shared
        }
        synchronized(...) { // CS2
            G.X = G.X + 1; // Read from X, thread-shared
            Write to X, thread-shared
        }
    }
}

class T2 extends Runnable {
    run() {
        synchronized(...) { // CS3
            int a = G.Y; // Read from Y, thread-shared
        }
    }
}
May Happen in Parallel Analysis

Find and apply MHP information

class G {
    public static int X, Y;
}

class T1 extends Runnable {
    run() {
        synchronized(...) { // CS1
            G.Y = G.X;
        }
        synchronized(...) { // CS2
            G.X = G.X + 1;
        }
    }
}

class T2 extends Runnable {
    run() {
        synchronized(...) { // CS3
            int a = G.Y;
        }
    }
}
Component-Based Lock Allocation

Static Lock Allocation:
(Dynamic is the same in this case)

```java
class G {
    public static int X, Y;
}

class T1 extends Runnable {
    run() {
        synchronized(LockObject1) { // CS1
            G.Y = G.X;
        }
        synchronized(...) { // CS2
            G.X = G.X + 1;
        }
    }
}

class T2 extends Runnable {
    run() {
        synchronized(LockObject1) { // CS3
            int a = G.Y;
        }
    }
    public static Object LockObject1 = new Object();
}
```
Finding Thread-Based Side Effects

Build on an existing side-effect analysis

- Identify fields that are read & written
- Each field has a points-to set of possible base objects

Extend it to be thread-sensitive

- Approximate the thread-visible effects of library calls
- Exclude thread-local side effects

Use it to construct a critical section *interference graph*
class A {
    public static int f;
    
    synchronized void a() {
        A.f = B.f + 1;
    }
}

class B {
    public static int f;
    
    synchronized void b() {
        B.f = B.f + D.f;
    }
}

class C {
    public static int f;
    
    synchronized void c() {
        C.f = C.f + 1;
    }
}

class D {
    public static int f;
    
    synchronized void d() {
        D.f = D.f + 1;
    }
}
Constructing an Interference Graph

Interference Graph

A B C D
A 1 1 0 0
B 1 1 0 1
C 0 0 1 0
D 0 1 0 1
**Finding Thread-Local Objects**

*Thread-local object*: object only read & written by a single thread

Similar to escape analysis

- Partition the heap into thread-shared and thread-local data
- Use information flow analysis to propagate thread-shared status

Values identified as thread-local do not require synchronized access
MHP analysis finds methods that execute concurrently

Several distinct steps:
1. Identify run-once and run-many statements
2. Identify run-once and run-many threads
3. Categorize run-many threads as run-one-at-a-time or run-many-at-time
4. Find methods that may happen in parallel based on thread reachability

Critical sections that may not happen in parallel cannot interfere!
Run-Once Run-Many Analysis
Run-Once Run-Many Analysis
Run-Once Run-Many Analysis

foo() -> bar() -> main()
Run-Once Run-Many Analysis

foo() -> bar() -> main()
Run-Once Run-Many Analysis
Thread t1, t2, t3

```
int i
```

t1 = new T1()
t1.start()
t2 = new T2()
i = 0
if (i > 0)

T1: ?
T2: ?
T3: ?

t2.start()
t3 = new T3()
t3.start()
i = i + 1
if (i < 10)

run-once
run-many
Thread t1, t2, t3

int i

- **run-once**
- **run-many**

T1: ?
T2: ?
T3: ?

t1 = new T1()
t1.start()
t2 = new T2()
i = 0
if (i > 0)
  t2.start()
t3 = new T3()
t3.start()
i = i + 1
if (i < 10)
  t3.start()
  t2.start()
  t1.start()
Thread t1, t2, t3
int i

- Blue: run-once
- Red: run-many

T1: run-once
T2: ?
T3: ?

t1 = new T1()
t1.start()
t2 = new T2()
i = 0
if (i > 0)
  t2.start()
t3 = new T3()
t3.start()
i = i + 1
if (i < 10)
Thread t1, t2, t3
int i

- **run-once**
- **run-many**

T1: run-once
T2: run-once
T3: ?

---

```
t1 = new T1()
t1.start()
t2 = new T2()
i = 0
if (i > 0)
  t3 = new T3()
t3.start()
i = i + 1
if (i < 10)
```

**Flowchart:**
- t2.start() (F)
- t3 = new T3() (T)
- t3.start() (T)
- i = i + 1
- if (i < 10) (T)
- if (i > 0) (T)

**Decision Points:**
- t2.start() (F)
- if (i > 0) (T, F)
- if (i < 10) (T, F)
Thread t1, t2, t3

```java
int i

// T1: run-once
int i = 0
if (i > 0) {
    t2.start();
    t2 = new T2();
}

// T2: run-once
if (i < 10) {
    t3 = new T3();
    t3.start();
    i = i + 1;
}

// T3: run-many
if (i < 10) {
    t3 = new T3();
    t3.start();
    i = i + 1;
}
```
Finding run-one-at-a-time threads

Thread t1, t2, t3
int i

T1: run-once
T2: run-once
T3: run-many
one-at-a-time

```java
Thread t1, t2, t3
int i

t1 = new T1()
t1.start()
t2 = new T2()
i = 0
if (i > 0)
    t3 = new T3()
    t3.start()
    t3.join()
i = i + 1
if (i < 10)
```
For each start, consider all joins:
Finding run-one-at-a-time threads

- For each start, consider all joins:
  - Any valid join receiver must alias start receiver
For each start, consider all joins:

- Any valid join receiver must alias start receiver
- Any valid join must post-dominate start
Finding run-one-at-a-time threads

For each start, consider all joins:

- Any valid join receiver must alias start receiver
- Any valid join must post-dominate start
- And not have loops to start between the start and join...
Finding run-one-at-a-time threads

For each start, consider all joins:
- Any valid join receiver must alias start receiver
- Any valid join must post-dominate start
- And not have loops to start between the start and join...

If join is valid, check method validity:
- Method must not be called recursively
- Method must not happen in parallel with itself
Finding MHP Information

run-once

run-one-at-a-time

run-many-at-a-time

MHP Information

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Applying MHP Information

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

A simple Hadamard product
Applying MHP Information

A simple Hadamard product
Three kinds of component-based lock allocation:

1. Singleton: a single static lock protects all components
2. Static: one static lock per component
3. Dynamic: attempt to use per-data structure locks for each component, otherwise static

Finally, isolated vertices with no self loops are *unlocked*
1 Introduction
2 Design
3 Experimental Results
4 Conclusions and Future Work
For each benchmark, we do 13 experiments:

- **control**: original benchmark program
- **singleton**: single static lock for all critical sections
- **5 static locking allocations**:
  - 1. **CHA**: class hierarchy analysis points-to and side effects
  - 2. **Spark**: context-insensitive points-to and side effects
  - 3. **Spark-MHP**: Spark with may happen in parallel [MHP] analysis
  - 4. **Spark-TLO-MHP**: Spark with both TLO and MHP
- **5 analogous dynamic locking allocations**

11 benchmarks: 5 micro, 6 standard

64-bit AMD Machines (dual, 4-way, 4-way dual), Sun JDK1.5
Relative Speedup of Using CHA

Performance Relative to Manual Allocation

slowdown

traffic  sync.methods  sync.objects  lusearch  jbb2000  jbb2005
Relative Speedup of Using Spark

Performance Relative to Manual Allocation

slowdown

traffic  sync.methods  sync.objects  lusearch  jbb2000  jbb2005

2  4  8  2  4  8  2  4  8  2  4  8

1  1  1  1  1  1
Relative Speedup of Adding TLO Analysis

Performance Relative to Manual Allocation

slowdown

2 4 8 traffic sync.methods sync.objects lusearch jbb2000 jbb2005
Relative Speedup of Using Dynamic Locking

Performance Relative to Manual Allocation

- traffic
- sync.methods
- sync.objects
- lusearch
- jbb2000
- jbb2005
Relative Speedup of Using Dynamic Locking

Performance Relative to Manual Allocation

slowdown
Outline

1. Introduction
2. Design
3. Experimental Results
4. Conclusions and Future Work
Conclusions

- Singleton allocation is not generally viable
- Points-to analysis precision is important
- MHP analysis helps if it can split a larger component
- TLO analysis usually has a negligible effect
- Dynamic locking has a small impact; may degrade or improve performance
- Component-based allocation works surprisingly well for many benchmarks
Future Work

- More precise compiler analyses
- Finer locking granularities
- Method synchronization
- Critical section inference
- Speculative locking and transactional memory
Thank you for your attention.
May Happen in Parallel analysis for Java (Naumovich et al. ’99, Li ’04).

Thread-sensitive points-to and escape analysis (Chang and Choi ’04, Sălcianu and Rinard ’01).

Thread-local objects analysis for synchronization elimination (Ruf ’00).

Pessimistic atomic sections/transactions (McCloskey et al. ’06, Hicks et al. ’06).

Lock allocation
  - Concurrency graph (Sreedhar, Zhang, et al. ’05).

Static race detection (Naik et al.’06, and many others).

Optimistic concurrency, transactional memory (see Larus & Rajwar ’06).