Making Sequential Consistency Practical in Titanium

Amir Kamil, Jimmy Su, and Katherine Yelick
Titanium Group
http://titanium.cs.berkeley.edu

U.C. Berkeley
November 15, 2005
Reordering in Sequential Programs

Two accesses can be reordered as long as the reordering does not violate a local dependency.

Initially, $\text{flag} = \text{data} = 0$

\[
\begin{array}{c}
\text{data} = 1 \\
\downarrow \\
\text{flag} = 1
\end{array}
\quad \rightarrow \quad
\begin{array}{c}
\text{flag} = 1 \\
\downarrow \\
\text{data} = 1
\end{array}
\]

In both orderings, the end result is \{data == flag == 1\}.
Reordering in Parallel Programs

In parallel programs, a reordering can change the semantics even if no local dependencies exist.

Initially, \( \text{flag} = \text{data} = 0 \)

\[
\begin{align*}
\text{T1} & : \quad \text{data} = 1 \\
& \quad \text{flag} = 1 \\
\text{T2} & : \quad f = \text{flag} \\
& \quad d = \text{data} \quad & \quad f = \text{flag} \\
\end{align*}
\]

\( \{ f == 1, d == 0 \} \) is a possible result in the reordered code, but not in the original code.
Memory Models

• In *relaxed consistency*, reordering is allowed if no local dependencies or synchronization operations are violated.

• In *sequential consistency*, a reordering is illegal if it can be observed by another thread.

• Titanium, Java, UPC, and many other languages *do not* provide sequential consistency due to the (perceived) cost of enforcing it.
Software and Hardware Reordering

- Compiler can reorder accesses as part of an optimization
  - Example: copy propagation
  - *Logical fences* inserted where reordering is illegal
    – optimizations respect these fences

- Hardware can reorder accesses
  - Examples: out of order execution, remote accesses
  - *Fence instructions* inserted into generated code – waits until all prior memory operations have completed
  - Can cost a complete round trip time due to remote accesses
Conflicts

• Reordering of an access is observable only if it conflicts with some other access:
  • The accesses can be to the same memory location
  • At least one access is a write
  • The accesses can run concurrently

Conflicts

• Fences only need to be inserted around accesses that conflict
Sequential Consistency in Titanium

- Minimize number of fences – allow same optimizations as relaxed model
- Concurrency analysis identifies concurrent accesses
  - Relies on Titanium’s textual barriers and single-valued expressions
- Alias analysis identifies accesses to the same location
  - Relies on SPMD nature of Titanium
Barrier Alignment

• Many parallel languages make no attempt to ensure that barriers line up
  • Example code that is legal but will deadlock:
    ```c
    if (Ti.thisProc() % 2 == 0)
        Ti.barrier(); // even ID threads
    else
        ; // odd ID threads
    ```
Structural Correctness

• Aiken and Gay introduced *structural correctness* (*POPL’98*)
  • Ensures that every thread executes the same number of barriers
  • Example of structurally correct code:
    
    ```java
    if (Ti.thisProc() % 2 == 0)
        Ti.barrier(); // even ID threads
    else
        Ti.barrier(); // odd ID threads
    ```
Textual Barrier Alignment

- Titanium has *textual barriers*: all threads must execute the same *textual* sequence of barriers
  - Stronger guarantee than structural correctness – this example is illegal:
    ```java
    if (Ti.thisProc() % 2 == 0)
        Ti.barrier(); // even ID threads
    else
        Ti.barrier(); // odd ID threads
    ```
- *Single-valued* expressions used to enforce textual barriers
**Single-Valued Expressions**

- A *single-valued* expression has the same value on all threads when evaluated
  - Example: `Ti.numProcs() > 1`
- All threads guaranteed to take the same branch of a conditional guarded by a single-valued expression
  - Only single-valued conditionals may have barriers
  - Example of legal barrier use:
    ```java
    if (Ti.numProcs() > 1)
        Ti.barrier(); // multiple threads
    else
        ; // only one thread total
    ```
Concurrent Analysis (I)

- Graph generated from program as follows:
  - Node added for each code segment between barriers and single-valued conditionals
  - Edges added to represent control flow between segments

```c
// code segment 1
if ([single])
  // code segment 2
else
  // code segment 3
// code segment 4
Ti.barrier()
// code segment 5
```
Concurrency Analysis (II)

• Two accesses can run concurrently if:
  • They are in the same node, or
  • One access’s node is reachable from the other access’s node without hitting a barrier

• Algorithm: remove barrier edges, do DFS

<table>
<thead>
<tr>
<th>Concurrent Segments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>
Alias Analysis

• Allocation sites correspond to abstract locations (a-locs)
• All explicit and implicit program variables have points-to sets
• A-locs are typed and have points-to sets for each field of the corresponding type
  • Arrays have a single points-to set for all indices
• Analysis is flow, context-insensitive
  • Experimental call-site sensitive version – doesn’t seem to help much
Thread-Aware Alias Analysis

• Two types of abstract locations: local and remote
  • Local locations reside in local thread’s memory
  • Remote locations reside on another thread

• Exploits SPMD property
  • Results are a summary over all threads
  • Independent of the number of threads at runtime
**Alias Analysis: Allocation**

- Creates new local abstract location
- Result of allocation must reside in local memory

```java
class Foo {
    Object z;
}

static void bar() {
    Foo a = new Foo();
    Foo b = broadcast a from 0;
    Foo c = a;
    a.z = new Object();
}
```

<table>
<thead>
<tr>
<th>A-locs</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points-to Sets</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Alias Analysis: Assignment

- Copies source abstract locations into points-to set of target

```java
class Foo {
    Object z;
}

static void bar() {
    Foo a = new Foo();
    Foo b = broadcast a from 0;
    Foo c = a;
    a.z = new Object();
}
```

### Points-to Sets

<table>
<thead>
<tr>
<th>A-locs</th>
<th>1, 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points-to Sets</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>1</td>
</tr>
<tr>
<td>1.z</td>
<td>2</td>
</tr>
</tbody>
</table>
Alias Analysis: Broadcast

• Produces both local and remote versions of source abstract location
  • Remote a-loc points to remote analog of what local a-loc points to

```java
class Foo {
    Object z;
}

static void bar() {
    L1: Foo a = new Foo();
    Foo b = broadcast a from 0;
    Foo c = a;
    L2: a.z = new Object();
}
```

<table>
<thead>
<tr>
<th>Points-to Sets</th>
<th>A-locs</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>1, 1_r</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
</tr>
<tr>
<td>1.z</td>
<td>2</td>
</tr>
<tr>
<td>1_r.z</td>
<td>2_r</td>
</tr>
</tbody>
</table>
Aliasing Results

- Two variables $A$ and $B$ may alias if:
  \[ \exists x \in \text{pointsTo}(A). \]
  \[ x \in \text{pointsTo}(B) \]

- Two variables $A$ and $B$ may alias across threads if:
  \[ \exists x \in \text{pointsTo}(A). \]
  \[ R(x) \in \text{pointsTo}(B), \]
  (where $R(x)$ is the remote counterpart of $x$)

<table>
<thead>
<tr>
<th>Points-to Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
</tr>
<tr>
<td>$b$</td>
</tr>
<tr>
<td>$c$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alias [Across Threads]:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
</tr>
<tr>
<td>$b$</td>
</tr>
<tr>
<td>$c$</td>
</tr>
</tbody>
</table>
## Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pi</td>
<td>56</td>
<td>Monte Carlo integration</td>
</tr>
<tr>
<td>demv</td>
<td>122</td>
<td>Dense matrix-vector multiply</td>
</tr>
<tr>
<td>sample-sort</td>
<td>321</td>
<td>Parallel sort</td>
</tr>
<tr>
<td>lu-fact</td>
<td>420</td>
<td>Dense linear algebra</td>
</tr>
<tr>
<td>3d-fft</td>
<td>614</td>
<td>Fourier transform</td>
</tr>
<tr>
<td>gsrb</td>
<td>1090</td>
<td>Computational fluid dynamics kernel</td>
</tr>
<tr>
<td>gsrb*</td>
<td>1099</td>
<td>Slightly modified version of gsrb</td>
</tr>
<tr>
<td>spmv</td>
<td>1493</td>
<td>Sparse matrix-vector multiply</td>
</tr>
<tr>
<td>gas</td>
<td>8841</td>
<td>Hyperbolic solver for gas dynamics</td>
</tr>
</tbody>
</table>

1 Line counts do not include the reachable portion of the 37,000 line Titanium/Java 1.0 libraries
**Analysis Levels**

- We tested analyses of varying levels of precision

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>naïve</td>
<td>All heap accesses</td>
</tr>
<tr>
<td>sharing</td>
<td>All shared accesses</td>
</tr>
<tr>
<td>concur</td>
<td>Concurrency analysis + type-based AA</td>
</tr>
<tr>
<td>concur/saa</td>
<td>Concurrency analysis + sequential AA</td>
</tr>
<tr>
<td>concur/taa</td>
<td>Concurrency analysis + thread-aware AA</td>
</tr>
<tr>
<td>concur/taa/cycle</td>
<td>Concurrency analysis + thread-aware AA + cycle detection</td>
</tr>
</tbody>
</table>
Static (Logical) Fences

Percentages are for number of static fences reduced over naive
Dynamic (Executed) Fences

Percentages are for number of dynamic fences reduced over naive.
**Dynamic Fences: gsrb**

- gsrb relies on dynamic locality checks
  - slight modification to remove checks (gsrb*) greatly increases precision of analysis

![]()
Two Example Optimizations

• Consider two optimizations for GAS languages
  1. Overlap bulk memory copies
  2. Communication aggregation for irregular array accesses (i.e. \(a[b[i]]\))

• Both optimizations reorder accesses, so sequential consistency can inhibit them

• Both are addressing network performance, so potential payoff is high
Array Copies in Titanium

- Array copy operations are commonly used
  
  ```javascript
  dst.copy(src);
  ```

- Content in the domain intersection of the two arrays is copied from `dst` to `src`

- Communication (possibly with packing) required if arrays reside on different threads

- Processor blocks until the operation is complete.
Non-Blocking Array Copy Optimization

• Automatically convert blocking array copies into non-blocking array copies
  • Push sync as far down the instruction stream as possible to allow overlap with computation

• Interprocedural: syncs can be moved across method boundaries

• Optimization reorders memory accesses – may be illegal under sequential consistency
Communication Aggregation on Irregular Array Accesses (Inspector/Executor)

- A loop containing indirect array accesses is split into phases
  - *Inspector* examines loop and computes reference targets
  - Required remote data gathered in a bulk operation
  - *Executor* uses data to perform actual computation

```plaintext
for (...) {
    a[i] = remote[b[i]];
    // other accesses
}
```

```plaintext
schd = inspect(remote, b);
tmp = get(remote, schd);
for (...) {
    a[i] = tmp[i];
    // other accesses
}
```

- Can be illegal under sequential consistency
**Relaxed + SC with 3 Analyses**

- We tested performance using analyses of varying levels of precision

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>relaxed</td>
<td>Uses Titanium’s relaxed memory model</td>
</tr>
<tr>
<td>naïve</td>
<td>Uses sequential consistency, puts fences around every heap access</td>
</tr>
<tr>
<td>sharing</td>
<td>Uses sequential consistency, puts fences around every shared heap access</td>
</tr>
<tr>
<td>concur/taa/cycle</td>
<td>Uses sequential consistency, uses our most aggressive analysis</td>
</tr>
</tbody>
</table>
Dense Matrix Vector Multiply

- Non-blocking array copy optimization applied
- Strongest analysis is necessary: other SC implementations suffer relative to relaxed
Sparse Matrix Vector Multiply

- Inspector/executor optimization applied
- Strongest analysis is again necessary and sufficient
Conclusion

• Titanium’s textual barriers and single-valued expressions allow for simple but precise concurrency analysis.

• Sequential consistency can eliminate nearly all fences for the benchmarks tested.

• On two linear algebra kernels, sequential consistency can be provided with little or no performance cost with our analysis.
  • Analysis allows the same optimizations to be performed as in the relaxed memory model.