Grace: Safe Multithreaded Programming for C/C++

Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. 2009

Speaker: Ivo Steinmann
Concurrent Computation

- Cyclic lock acquisition → deadlocks
- Unguarded update → race conditions
- Unguarded, interleaved updates → atomicity violations
- Threads scheduled in unexpected order → order violations
Solution: Grace

- Locks converted to no-ops
  \[\text{deadlocks}\]
- All updates committed deterministically (sequential)
  \[\text{race-conditions}\]
- Threads run atomically
  \[\text{atomicity-violations}\]
- Threads execute in program order (sequential)
  \[\text{order-violations}\]
Sequential Semantics (1)

- Restricted to fork-join parallelism

// Run f(x) and g(y) in parallel
T1 = spawn f(x);
T2 = spawn g(y);
// Wait for both to complete
sync;
Sequential Semantics (2)

- Program is *behaviorally* turned into its sequential counterpart → serial elision

```c
// Run f(x) and g(y) in parallel.
t1 = spawn f(x);
t2 = spawn g(y);
// Wait for both to complete
sync;
```

- thread spawn → sequential
- lock operations → no-ops
Sequential Semantics (3)

- Threads run concurrently
- Committed in sequential order
  - Each thread waits for its logical predecessor

```plaintext
// Run f(x) and g(y) in parallel.
t1 = spawn f(x);
t2 = spawn g(y);
// Wait for both to complete
sync;
```

Transactional Memory required

```

```

Transational Memory required

```
Transactional Memory (1)

- Updates are committed in program order
  - Some form of transactional memory required
  - \textit{atomic} clause for short transactions
  - What about long-lived transactions?

- Solution: use processes instead of threads (\texttt{forks})
  - Standard memory protection functions
  - Signal handlers to track reads/writes
  - Shared address space
Transactional Memory (2)

- Memory mapped files → shared memory
  - Array of version numbers (one per page)
  - Shared mapping → latest commit state
  - Local (per-process), copy-on-write mapping → working set
Transaction Memory (3)

- Local mapping
  - Protection of all pages set to PROT_NONE
  - First access triggers a pagefault (SEGV)
    - Read: set protection to PROT_READ
    - Write: set protection to PROT_READ | PROT_WRITE and update version
  - Copy on write semantics!
Transactional Memory (4) - commit

1. thread (process) begin
2. read page 1
3. read page 4
4. write page 4
5. thread end
6. wait for logical predecessor
7. consistency checks
8. commit

Committed (shared) pages

Uncommitted (private) pages

Protected
Read-only
Unprotected
Transactional Memory (5) - rollback

1. thread (process) begin
2. read page 1
3. read page 4
4. write page 4
5. thread end
6. wait for logical predecessor
7. consistency checks
8. rollback → reexecute
Thread Execution (1)

- Initialization
  - Save execution context
    - program counter
    - registers
    - stack contents
  - Set page protection to PROT_NONE

- Execution
  - Track page accesses over SEGV protection faults
  - Version control
Thread Execution (2)

- **Completion**
  - Commit attempts at
    - end of main()
    - end of individual thread
    - right before a child thread spawn
    - right before joining another thread
  - **No commit required when no change**
Thread Execution (3)

- Committing
  - lock all memory mappings (interprocess mutex)
  - perform consistency checks (check version numbers)
    - success: copy contents of each page into shared images
    - fail: rollback and reexecute
Benchmarks

- **histogram** Analyzes images' RGB components
- **kmeans** Iterative clustering of 3-D points
- **linear_regression** Computes best fit line for a set of points
- **matmul** Recursive matrix-multiply
- **pca** Principal component analysis on matrix
- **string_match** Searches file for encrypted word
## Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Commits</th>
<th>Rollbacks</th>
<th>Pages Read</th>
<th>Pages Written</th>
</tr>
</thead>
<tbody>
<tr>
<td>histogram</td>
<td>9</td>
<td>0</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>kmeans</td>
<td>6273</td>
<td>4887</td>
<td>404</td>
<td>2</td>
</tr>
<tr>
<td>linear_reg</td>
<td>9</td>
<td>0</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>matmul</td>
<td>11</td>
<td>0</td>
<td>4100</td>
<td>1865</td>
</tr>
<tr>
<td>pca</td>
<td>22</td>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>string_match</td>
<td>11</td>
<td>0</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

![Graph showing speedup comparison between pthreads and Grace]
Verification – Deadlocks

```c
thread1() {
    lock(A);
    lock(B);
    // ...do something
    unlock(B);
    unlock(A);
}

thread2() {
    lock(B);
    lock(A);
    // ...do something
    unlock(A);
    unlock(B);
}
```
Verification – Atomicity violations

```c
thread1() {
    if (thd->proc_info) {
        fputs(thd->proc_info, ...);
    }
}

thread2() {
    thd->proc_info = NULL;
}
```
Verification – Race conditions

```c
int counter = 0;

increment() {
    print(counter);
    int temp = counter;
    temp++;
    counter = temp;
    print(counter);
}

thread1() { increment(); }
thread2() { increment(); }
```
Verification – Order violations

```c
char* proc_info;

thread1() {
    proc_info = malloc(256);
}

thread2() {
    // maybe executed before thread1()
    strcpy(proc_info, "abc");
}

main() {
    spawn thread1();
    spawn thread2();
}
```
Questions?