Deterministic Execution

02/27 Ni Kang, Taihai He
DMP: deterministic shared memory multiprocessing

- Summary
- DMP Key Mechanisms
  - DMP-Serial
    - Deterministic Token and Quantum
  - DMP-ShTab
  - DMP-TM/DMP-TMFwd
  - Quantum Builders
- Hardware/Sofeware Implementations
- Evaluations
- Discussions
Background

- Nondeterminism
  - Same inputs can lead to different outputs
  - Too many possible ways of instruction interleaving
    - “Defective software might execute correctly hundreds of times before a subtle synchronization bug appears, and when it does, developers typically cannot reproduce it during debugging.”
  - Need to use logs to record every execution
    - Still hard to replay
Summary

- **Determinism**
  - Key: deterministic inter-thread communication
    - Maintain a fixed order of load/store operations on shared data
    - Rest of the instructions can still have different orders in executions
      - “Communication-equivalent interleavings”
  - Use deterministic execution to improve reliability
    - Easier to test and debug
      - Avoid subtle multithread bugs
      - Always able to reproduce previous execution results
    - Acceptable performance loss
      - Multiple co-existable mechanisms for different applications
      - Complexity-performance trade-offs between hardware and software implementations
Nondeterminism Quantification

- Exist regions where nondeterminism drops to nearly zero.
- Executions may never reach 100% nondeterminism.

Figure 3. Amount of nondeterminism over the execution of barnes and ocean-contig. The x axis is the position in the execution where each sample of 100,000 instructions was taken. The y axis is $ND$ (Eq. 1) computed for each sample in the execution.
**DMP-Serial**

- Fully serialized accesses to data
- Allow only one processor at a time to access memory in a deterministic order

---

Figure 4. Deterministic serialization of memory operations. Dots are memory operations and dashed arrows are happens-before synchronization.
Deterministic Token and Quantum

- Deterministic token
  - Processor with the token can access memory. Otherwise, wait for it.
  - One token passing around. Multiple tokens are also allowed with hardware implementation if there are multiple deterministic processes at the same time

- Quantum
  - Instruction segments involving shared data load/store that require token
  - QB-Count: count instructions and break when a deterministic, target number is reached
  - Other smarter ways to divide quantum, will introduce later

- Parallelism?
  - Serialization hurts performance a lot
DMP-ShTab

- Not all load/store operations have conflicts
  - Communication is the key
  - Quantum = communication-free prefix + serial suffix
  - Only requires suffixies to be deterministic

- Sharing table for memory locations
  - Data is either private or shared for a processor
  - Supports different granularities

- Features
  - Token is only required for accessing shared data
  - If one thread wants to write data, it needs to wait for all other threads to be blocked even if it has already acquired the token. (Broadcast)
    - Block: finish execution of quantum or prefix

Figure 6. Deterministic serialization of shared memory communication only.
DMP-TM / DMP-TMFwd

- Transactional Memory Support
  - Allowing more concurrent executions with speculations and re-executions

- DMP-TM
  - Speculation + Commit + Squash
  - Correctness: no overlapping memory accesses
  - May squash and re-execute quantum when deterministic serialization is violated

- DMP-TMFwd
  - DMP-TM + Forward
  - Quantum can fetch uncommitted data from other quantum
  - Avoid some squashes, but all subsequent quantum need to be squashed if previous speculations generated incorrect data
Quantum Builders

- A fixed number of instructions may not reflect the progress of a thread on its critical path of execution
  - QB-SyncFollow
    - Ends a quantum when an unlock operation is performed
    - Other threads may be waiting for the lock right now
  - QB-Sharing
    - Ends a quantum when a thread hasn’t issued memory operations to shared locations in some time, like after a number of instructions
    - Other threads don’t need to keep waiting if current thread has already finished all of its memory-sensitive operations
- QB-SyncSharing
  - QB-SyncFollow OR QB-Sharing, whenever either of their requirements are satisfied
Hardware / Software Implementation

- Hardware: more complex, better performance (less performance drop)
  - Quantum Building: may need supports from compilers
  - DMP-ShTab
    - Uses MESI cache coherence protocol to represent private / shared status
    - State changing requirements: no speculation, must have token, all threads blocked
    - Similar to directory-based cache coherence
  - DMP-TM / DMP-TMFwd
    - Allowing commit only when token is held
    - Data versioning
    - Similar to Thread-Level Speculation (TLS)
- Software: simple, helpful at debugging-level
  - Use compiler or binary writer
  - Build quantum with CFG
  - Token = lock
Evaluation: mechanisms

Figure 9. Runtime overheads with 4, 8 and 16 threads. (P) indicates page-level conflict detection; line-level otherwise.
Evaluation: quantum size

Figure 10. Performance of 2,000 (2), 10,000 (X) and 100,000 (C) instruction quanta, relative to 1,000 instruction quanta.
Evaluation: granularity

Figure 11. Performance of page-granularity conflict detection, relative to line-granularity.
Evaluation: quantum builders

Figure 12. Performance of QB-Sharing (s), QB-SyncFollow (sf) and QB-SyncSharing (ss) quantum builders, relative to QB-Count, with 1,000-instruction quanta.

Figure 13. Performance of quantum building schemes, relative to QB-Count, with 10,000-instruction quanta.
## Evaluation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Hw-DMP Implementation – 1,000-instr quanta</th>
<th>QB Strategy – 10,000-instr quanta†</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TM</td>
<td>SyncFollow</td>
</tr>
<tr>
<td>barnes</td>
<td>27/9</td>
<td>37</td>
</tr>
<tr>
<td>cholesky</td>
<td>14/6</td>
<td>23</td>
</tr>
<tr>
<td>fft</td>
<td>22/16</td>
<td>25</td>
</tr>
<tr>
<td>fmm</td>
<td>30/6</td>
<td>51</td>
</tr>
<tr>
<td>lu-nc</td>
<td>47/33</td>
<td>71</td>
</tr>
<tr>
<td>ocean-c</td>
<td>46/15</td>
<td>28</td>
</tr>
<tr>
<td>radix</td>
<td>16/20</td>
<td>7</td>
</tr>
<tr>
<td>vldrmd</td>
<td>27/8</td>
<td>38</td>
</tr>
<tr>
<td>water-sp</td>
<td>32/19</td>
<td>19</td>
</tr>
<tr>
<td>SPLASH amean</td>
<td>30/16</td>
<td>31</td>
</tr>
<tr>
<td>blacksch</td>
<td>28/9</td>
<td>8</td>
</tr>
<tr>
<td>bodytr</td>
<td>11/4</td>
<td>16</td>
</tr>
<tr>
<td>fluid</td>
<td>41/8</td>
<td>76</td>
</tr>
<tr>
<td>strmcl</td>
<td>36/5</td>
<td>28</td>
</tr>
<tr>
<td>PARSEC amean</td>
<td>29/6</td>
<td>36</td>
</tr>
</tbody>
</table>

Table 1. Characterization of hardware-DMP results. † Same granularity as used in Figure 9
Evaluation:
software implementation

Figure 14. Runtime of Sw-DMP-ShTab relative to nondeterministic execution.
Discussions

- A system can have DMP-TM(Fwd) / DMP-ShTab / DMP-Serial at the same time and switch to each other for different tasks
- Hardware and software implementations can be used together to have flexibility
- Supports deployment with modification and standardization
Grace: Safe Multithreaded Programming for C/C++
Motivation

- Concurrency bugs

<table>
<thead>
<tr>
<th>Concurrency Error</th>
<th>Cause</th>
<th>Prevention by Grace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadlock</td>
<td>cyclic lock acquisition</td>
<td>locks converted to no-ops</td>
</tr>
<tr>
<td>Race condition</td>
<td>unguarded updates</td>
<td>all updates committed deterministically</td>
</tr>
<tr>
<td>Atomicity violation</td>
<td>unguarded, interleaved updates</td>
<td>threads run atomically</td>
</tr>
<tr>
<td>Order violation</td>
<td>threads scheduled in unexpected order</td>
<td>threads execute in program order</td>
</tr>
</tbody>
</table>

Table 1. The concurrency errors that Grace addresses, their causes, and how Grace eliminates them.
Motivation

- Transactional memory system is not working here
- Compatibility with C/C++ and commodity hardware
- Support for long-lived transactions
- Isolation of updates from other threads
- Support for irrevocable actions (i.e. I/O)
- Low runtime and space overhead
Introduction

- Treating threads as processes
  - Use memory mapped files to share the heap and globals across processes
  - Version numbers
Introduction

- Globals
- Heap Organization
  - Fixed size heap
  - Sub-heap
Execution -- Initialization

```c
void atomicBegin (void) {
    // Roll back to here on abort.
    // Saves PC, registers, stack.
    context.commit();
    // Reset pages seen (for signal handler).
    pages.clear();
    // Reset global and heap protection.
    globals.begin();
    heap.begin();
}
```

**Figure 4.** Pseudo-code for atomic begin.
Execution

committed (shared) pages & version numbers

uncommitted (private) pages

thread begin

protected
read-only
unprotected (copy-on-write)

thread end

reads
writes

{} {}

{} {}

{1} {}

{1,4} {}

{1,4} {4}
Execution -- Committing

- Locks are needed (mapping files)
- If version numbers for every page in the read set match the committed versions → Commit
- Else → Rollback
Sequential Commit

- Post-order traversal

Postorder Traversal: 4 5 2 6 7 3 1
Evaluation -- Concurrency Errors

- Deadlocks

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

thread2 () {
    lock (B);
    // usleep();
    lock (A);
    // ...do something
    unlock (A);
    unlock (B);
}
```
Evaluation -- Concurrency Errors

- Race conditions

```c
// Race condition.
int counter = 0;

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

thread1() { increment(); }
thread2() { increment(); }
```
Evaluation -- Concurrency Errors

- Atomicity violations

```c
// Atomicity violation.
// thread1
S1: if (thd->proc_info) {
   // usleep();
S2:   fputs (thd->proc_info,..)
      }

// thread2
S3: thd->proc_info = NULL;
```
Evaluation -- Concurrency Errors

- Order violations

```c
// Order violation.
char * proc_info;

thread1() {
    // ...
    // usleep();
    proc_info = malloc(256);
}

thread2() {
    // ...
    strcpy(proc_info,"abc");
}

main() {
    spawn thread1();
    // usleep();
    Spawn thread2();
    ...
    assert (foo == 0);
}
```
## Evaluation -- Real Applications

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Commits</th>
<th>Rollbacks</th>
<th>Pages Read</th>
<th>Pages Written</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>histogram</td>
<td>Analyzes images’ RGB components</td>
<td>9</td>
<td>0</td>
<td>7.3</td>
<td>5.9</td>
<td>1512.3</td>
</tr>
<tr>
<td>kmeans</td>
<td>Iterative clustering of 3-D points</td>
<td>6273</td>
<td>4887</td>
<td>404.5</td>
<td>2.3</td>
<td>8.7</td>
</tr>
<tr>
<td>linear_regression</td>
<td>Computes best fit line for set of points</td>
<td>9</td>
<td>0</td>
<td>5.6</td>
<td>4.8</td>
<td>1024.0</td>
</tr>
<tr>
<td>matmul</td>
<td>Recursive matrix-multiply</td>
<td>11</td>
<td>0</td>
<td>4100</td>
<td>1865</td>
<td>2359.4</td>
</tr>
<tr>
<td>pca</td>
<td>Principal component analysis on matrix</td>
<td>22</td>
<td>0</td>
<td>3.1</td>
<td>2.2</td>
<td>0.204</td>
</tr>
<tr>
<td>string_match</td>
<td>Searches file for encrypted word</td>
<td>11</td>
<td>0</td>
<td>5.9</td>
<td>4.3</td>
<td>191.1</td>
</tr>
</tbody>
</table>

*Table 2. CPU-intensive multithreaded benchmark suite and detailed characteristics (see Section 5.1).*
Evaluation -- Real application

- Thread-creation hoisting / argument padding
- Page-size base case
- Changed concurrency structure
Evaluation -- Real application

![CPU-intensive benchmarks](chart.png)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>pthreads</th>
<th>Grace</th>
</tr>
</thead>
<tbody>
<tr>
<td>histogram</td>
<td>5.8</td>
<td>4.8</td>
</tr>
<tr>
<td>kmeans</td>
<td>3.2</td>
<td>2.2</td>
</tr>
<tr>
<td>linear_regression</td>
<td>7.8</td>
<td>6.8</td>
</tr>
<tr>
<td>matmul</td>
<td>6.0</td>
<td>5.0</td>
</tr>
<tr>
<td>pca</td>
<td>12.9</td>
<td>10.8</td>
</tr>
<tr>
<td>string_match</td>
<td>6.0</td>
<td>5.0</td>
</tr>
</tbody>
</table>
Evaluation -- Application Characteristics

- Grain size

Figure 10. Impact of thread running time on performance: (a) speedup over a sequential version (higher is better), (b) normalized execution time with respect to \( p \) threads (lower is better).
Evaluation -- Application Characteristics

- Footprint

Figure 11. Impact of thread running time on performance: (a) speedup over a sequential version (higher is better), (b) normalized execution time with respect to pthread (lower is better).
Evaluation -- Application Characteristics

- Conflict rate

**Figure 12.** Impact of conflict rate (the likelihood of conflicting updates, which force rollbacks), versus a pthreads baseline that never rolls back (higher is better).
Thank you!