CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution

Tom Bergan
Owen Anderson, Joe Devietti, Luis Ceze, Dan Grossman
A Multithreaded Program

Thread 1

\[
t := x \\
x := t + 1
\]

Thread 2

\[
t := x \\
x := t + 1
\]

What is x?

\[
\begin{align*}
x & = 2 \\
x & = 2 \\
x & = 1
\end{align*}
\]
A Multithreaded Program

data race

Thread 1
atomicty violation

Thread 2
\{ 
\begin{align*}
t & := x \\
x & := t + 1 
\end{align*}
\}

What is $$x$$?

We’re not trying to make these bugs go away

We’re trying to make them come back!
Another Multithreaded Program

global $x=0$

**Thread 1**

```plaintext
lock(L)
assert(x!=42)
unlock(L)
```

**Thread 2**

```plaintext
lock(L)
x := 42
unlock(L)
```

no bug

BUG!
The Problem With Multithreading

- Shared-memory access interleavings are a hidden source of nondeterminism
  - hard to test
  - hard to debug
  - hard to replicate
Determinism Can Help!

hard to test
✓ test inputs, not interleavings
✓ software behaves as tested

hard to debug
✓ no more heisenbugs!
✓ reproduce bugs from the field

hard to replicate
✓ easy to synchronize replicas
Deterministic MultiProcessing

Goal: deterministic execution ...
• of arbitrary multithreaded programs
• without sacrificing scalability

Eliminate shared-memory nondeterminism
• execution is a function of inputs (including I/O)

**DMP [prior work, ASPLOS’09]:**
• hardware architecture for determinism
• using ownership-tracking and transactions
CoreDet: deterministic execution ...  
• of arbitrary, unmodified C/C++ pthreads programs  
• **without special hardware**  
• without sacrificing scalability
CoreDet: deterministic execution ...
• of arbitrary, unmodified C/C++ pthreads programs
• without special hardware
• without sacrificing scalability
CoreDet: deterministic execution ...
• of arbitrary, unmodified C/C++ pthreads programs
• without special hardware
• without sacrificing scalability

Contributions:
• new algorithm for deterministic execution
  ‣ uses store-buffering and relaxed memory consistency
    • compiler (LLVM pass) and a runtime library
      ‣ static optimizations
      ‣ dealing with external code
## Related Work

<table>
<thead>
<tr>
<th>Helps with . . .</th>
<th>Record + Replay</th>
<th>Kendo</th>
<th>DMP</th>
<th>CoreDet</th>
</tr>
</thead>
<tbody>
<tr>
<td>. . . testing?</td>
<td>○</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>. . . debugging?</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>. . . replication?</td>
<td>○ ●</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Assumes Race Free?

- Sometimes

### Needs hw?

- Usually
- No
- Yes
- No

### Examples:

- FDR, Rerun, Respec
- [ASPLOS’09] [ASPLOS’09] [ASPLOS’10]
Outline

Recap of DMP [ASPLOS’09]:

DMP-Ownership

DMP-TM

What’s wrong with doing these in software?

CoreDet:

- less complexity than DMP-TM
- with comparable scalability

not sequentially consistent!

Performance Evaluation
DMP-Serial [ASPLOS’09]

Thread 1

a := x
b := y
x := a * 2
y := a + b

Thread 2

c := x
d := y
x := a * 3
y := a - b
Thread 1

\[
\begin{align*}
a & := x \\
b & := y \\
x & := a \times 2 \\
y & := a + b
\end{align*}
\]

Thread 2

\[
\begin{align*}
c & := x \\
d & := y \\
x & := a \times 3 \\
y & := a - b
\end{align*}
\]
DMP-Serial [ASPLOS’09]

Execution is completely serialized
To recover parallelism ...  
... must resolve conflicts deterministically

by partitioning ownership (DMP-Ownership)

by using transactions (DMP-TM)
### DMP-Ownership

#### Parallel mode:
- No communication (can write only to private data)

#### Serial mode:
- Arbitrary communication

<table>
<thead>
<tr>
<th>MOT</th>
<th>Parallel</th>
<th>Serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>owned-by T&lt;sub&gt;1&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>shared</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>owned-by T&lt;sub&gt;2&lt;/sub&gt;</td>
<td>owned-by T&lt;sub&gt;3&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

- time →
- end of round
Start with DMP-Serial, then add transactions . . .
Execution is parallel and transactional
DMP in software

Can we implement DMP-Ownership in CoreDet?
✓ yes (we have!)
✗ sub-optimal scalability
   (too conservative about what can run in parallel)

Can we implement DMP-TM in CoreDet?
✗ not efficiently
   why not use STM?
What’s wrong with STM?

DMP-TM breaks important STM assumptions, specifically . . .

1) Transactions are rare
2) Transactions are short
3) Transactions are scoped

```c
void foo() {
  ...
  begin_transaction()
  return
}
```

An unscoped transaction:
Speculation makes things hard

Good scalability by allowing parallel updates of versioned memory (private transaction buffers)

CoreDet’s Insight:
Enable parallel updates without requiring speculation
Outline

Recap of DMP [ASPLOS’09]:

- DMP-Ownership
- DMP-TM

What’s wrong with DMP in software?

CoreDet:

- DMP-Buffering

Performance Evaluation
DMP-Buffering

Parallel Commiferial

T_1
T_2
T_3

time →

end of round
Parallel mode: buffer all stores (no communication)
**Parallel mode:** buffer all stores (no communication)

**Commit mode:** deterministically publish store buffers
Parallel mode: buffer all stores (no communication)
Commit mode: deterministically publish store buffers
Serial mode: used for synchronization (e.g. atomic ops)
**Parallel mode:** buffer stores locally

- ends at synchronization (atomic ops and fences), and quantum boundaries

**Commit mode:** publish local store buffers

- logically serial for determinism
- executes in parallel for performance

**Serial mode:** used for synchronization (e.g. atomic ops)
DMP-Buffering

Thread 1

\[ A = 1 \]
\[ \text{if } (B == 0) \]
\[ \ldots \]

Thread 2

\[ B = 1 \]
\[ \text{if } (A == 0) \]
\[ \ldots \]

Dekker’s Algorithm

(there is a data race)
Thread 1

buffer[A] = 1
if (B == 0)
...
A = buffer[A]

Thread 2

buffer[B] = 1
if (A == 0)
...
B = buffer[B]

DMP-Buffering
This is deterministic...
DMP-Buffering

Thread 1

buffer[A] = 1
if (B == 0)
...

A = buffer[A]

Thread 2

buffer[B] = 1
if (A == 0)
...

B = buffer[B]

...but not sequentially consistent
(cycle in the happens-before graph)
DMP-Buffering

Thread 1

\[ A = 1 \]
\[ tmp_1 = B \]

if (tmp\(_1\) == 0)

...

Thread 2

\[ B = 1 \]
\[ tmp_2 = A \]

if (tmp\(_2\) == 0)

...

Dekker’s Algorithm (again)
Let’s remove the data race …
DMP-Buffering

Thread 1

lock(L)

A = 1

tmp₁ = B

unlock(L)

if (tmp₁ == 0)
...

Thread 2

lock(L)

B = 1

tmp₂ = A

unlock(L)

if (tmp₂ == 0)
...

Dekker’s Algorithm
(no data race)
DMP-Buffering

lock(L)

A = 1
tmp₁ = B

unlock(L)

if (tmp₁ == 0)
...

lock(L)

B = 1
tmp₂ = A

unlock(L)

if (tmp₂ == 0)
...

Synchronization happens sequentially

serial

parallel + commit

serial

parallel + commit

serial

parallel + commit
lock(L)

A = 1
tmp₁ = B

unlock(L)

if (tmp₁ == 0)
...

lock(L)

B = 1
tmp₂ = A

unlock(L)

if (tmp₂ == 0)
...

Synchronization is a full fence
DMP-Buffering

lock(L)

A = 1

tmp₁ = B

unlock(L)

if (tmp₁ == 0)
...

Synchronization is a full fence

lock(L)

B = 1

tmp₂ = A

serial

parallel + commit

serial

parallel + commit

Data race free programs are sequentially consistent (required by C++ and Java memory models)
For determinism, the commit order must be deterministic 
\textit{i.e.} logically serial

For performance, the commit must happen in parallel

Basic idea:
• Publish store buffers in parallel
• Preserve the commit order on collisions
Basic idea:
• Publish store buffers in parallel
• Preserve the commit order on collisions
Basic idea:
• Publish store buffers in parallel
• Preserve the commit order on collisions

Detecting collisions
• Keep global record of published locations
• Locks to serialize writes

- Bloom filter to reduce locking overhead
Outline

Recap of DMP [ASPLOS’09]:

DMP-Ownership
DMP-TM

What’s wrong with DMP in software?

CoreDet:

DMP-Buffering

Performance Evaluation
Three algorithms implemented in CoreDet:

DMP-Ownership

DMP-Buffering

DMP-PartialBuffering
- a hybrid of DMP-Ownership and DMP-Buffering
- decides dynamically which data to buffer
Experimental Methodology

PARSEC and SPLASH2 benchmark suites

8-core Intel Xeon

scaled inputs to run for about a minute

Goal: in comparison to nondeterministic execution ... 

What is the scalability?

What are the overheads?
Scalability

Speedup over same strategy with 2 cores

2.4x speedup

splash mean

parsec mean
Since we preserve scalability, we can overcome overheads by adding cores.
CoreDet
• guarantees determinism in software of arbitrary C/C++ multithreaded programs

DMP-Buffering
• uses a relaxed memory consistency model
• scales comparably to nondeterministic execution
Also in the paper ...

Compiler details
• static optimizations
• forming balanced quanta

Runtime details
• dealing with external libraries
• threading libraries
• memory allocation

Evaluation
• more detailed performance characterization
Thank you!

Questions?

the CoreDet source code is available at
http://sampa.cs.washington.edu
(backup slides)
Atomic ops must happen in serial mode

\[ \text{CAS}(X, a, b) \quad \text{CAS}(X, a, c) \]
Atomic ops must happen in serial mode

\[
\begin{align*}
\text{tmp} &= x \\
\text{if} \ (\text{tmp} == a) \\
\text{x} &= b
\end{align*}
\]

\[
\begin{align*}
\text{tmp} &= x \\
\text{if} \ (\text{tmp} == a) \\
\text{x} &= c
\end{align*}
\]
Atomic ops must happen in serial mode

\[
\begin{align*}
\text{tmp} &= x \\
\text{if (tmp == a)} &\quad \text{tmp} &= x \\
\text{x} &= b &\quad \text{x} &= c
\end{align*}
\]

Synchronization, e.g. `lock()`, must happen in serial mode

- These are atomic ops
- There is an implied fence (must flush store buffers)
CoreDet: Implementation

A compiler (LLVM pass)
- instruments the code with calls to the runtime
- static optimizations to remove instrumentation
  - escape analysis
  - redundancy analysis

A runtime library
- scheduling threads
- tracks interthread communication
- deterministic wrappers for . . .
  - pthreads
  - malloc
Quantum Formation

“Just” instruction counting

Tension between:

- Perfect counting, for maximal balance
  - e.g. every basic block

- Minimal counting, for minimal overhead
  - e.g. only backedges and recursive calls

Heuristic compromise:

\[
\begin{align*}
3 & : cnt += 3 \\
5 & : cnt += 5 \\
50 & : cnt += 50
\end{align*}
\]
Remove Instrumentation From ...

• Accesses to thread-local (non-escaping) objects

• Redundant accesses

\[
y = \ldots x \ldots \\
z = \ldots x \ldots
\]

don’t need to instrument this
• Accesses to thread-local (non-escaping) objects

* **DMP-Buffering:** requires unification-based points-to analysis

```c
int local;
int *p = (...)? &local : &global;
...
```

must access through the store buffer

• Redundant accesses

```
y = ... x ...
z = ... x ...
```

don’t need to instrument this
• Accesses to thread-local (non-escaping) objects

*DMP-Buffering*: requires unification-based points-to analysis

```c
int local;
int *p = (...) ? &local : &global;
...
```

must access through the store buffer

• Redundant accesses

```c
y = ... x ...
z = ... x ...
```
don’t need to instrument this
Remove Instrumentation From ...

• Accesses to thread-local (non-escaping) objects

  **DMP-Buffering**: requires unification-based points-to analysis

  ```
  int local;
  int *p = (...) ? &local : &global;
  ...
  ```

  must access through the store buffer

• Redundant accesses

  ```
  y = ... x ...  
  z = ... x ...
  ```

  **DMP-Buffering**: this does not apply
External Libraries

We do not instrument external shared libraries, such as the system `libc`.

External calls must be serialized

Preventing over-serialization:
• We check indirect calls at runtime
• We provide deterministic wrappers for common `libc` functions, e.g. `memcpy` and `malloc`
• We do not serialize pure `libc` functions, e.g. `sqrt`