DeNovoSync: Efficient Support for Arbitrary Synchronization without Writer-Initiated Invalidations

Hyojin Sung and Sarita Adve

Department of Computer Science
University of Illinois, EPFL
Motivation

Complex software
Data races, non-determinism, implicit communication, ...

Shared Memory

Complex, inefficient hardware
Complex coherence, consistency, unnecessary traffic, ...
Motivation

Complex software
Data races, non-determinism, implicit communication, ...

WILD Shared Memory

Complex, inefficient hardware
Complex coherence, consistency, unnecessary traffic, ...
Motivation

Complex software
Data races, non-determinism, implicit communication, ...

Disciplined Shared Memory

Complex, inefficient hardware
Complex coherence, consistency, unnecessary traffic, ...
Motivation

Complex software
Data races, non-determinism, implicit communication, ...

Structured synch + Explicit memory side effects
Disciplined Shared Memory

Complex, inefficient hardware
Complex coherence, consistency, unnecessary traffic, ...
Motivation

Simpler

Complex software

No data races, safe non-determinism, explicit sharing, ...

Structured synch + Explicit memory side effects

Disciplined Shared Memory

Complex, inefficient hardware

Complex coherence, consistency, unnecessary traffic, ...
Motivation

Simpler

Complex software

No data races, safe non-determinism, explicit sharing, ...

Structured synch + Explicit memory side effects

Disciplined Shared Memory

Complex, inefficient hardware

Complex coherence, consistency, unnecessary traffic, ...

**Motivation**

**Simpler, more efficient**
Complex, inefficient hardware

- No data races, safe non-determinism, explicit sharing, ...
- Structured synch + Explicit memory side effects
- Disciplined Shared Memory

DeNovo [PACT11], DeNovoND [ASPLOS13, Top Picks 14]

BUT focus on data accesses, **synchronization** restricted
BUT much software (runtime, OS, ...) uses unstructured synch
Motivation

- Simpler
- Complex software
- No data races, safe non-determinism, explicit sharing, ...

DeNovoSync:
Support arbitrary synchronization with advantages of DeNovo

DeNovo [PACT11], DeNovoND [ASPLOS13, Top Picks 14]

BUT focus on data accesses, synchronization restricted
BUT much software (runtime, OS, ...) uses unstructured synch
Supporting Arbitrary Synchronization: The Challenge

- MESI: Writer sends invalidations to cached copies to avoid stale data

► Directory storage, inv/ack msgs, transient states, ...

Read A

↓

Write A

BUT Synchronization?
Naïve: Don’t cache synch

- Prior DeNovo assumptions
  - Race freedom
  - Restricted synchronization with special hardware

⇒ Reader self-invalidates stale data
Contributions of DeNovoSync

- **DeNovoSync**: Cache arbitrary synch w/o writer invalidations
- **Simplicity, perf, energy advantages of DeNovo w/o sw restrictions**
- **DeNovoSync vs. MESI for 24 kernels (16 & 64 cores), 13 apps**
  - Kernels: 22% lower exec time, 58% lower traffic for 44 of 48 cases
  - Apps: 4% lower exec time, 24% lower traffic for 12 of 13 cases
Outline

- Motivation
- **Background: DeNovo Coherence for Data**
- DeNovoSync Design
- Experiments
- Conclusions
DeNovo Coherence for Data (1 of 2)

• Original DeNovo software assumptions [PACT’11]
  – Data-race-free
  – Synchronization: Barriers demarcate parallel phases
  – Writeable data regions in parallel phase are explicit

• Coherence
  – Read hit: Don’t return stale data
    • Before next parallel phase, cache selectively self-invalidate
    – Needn’t invalidate data it accessed in previous phase
  – Read miss: Find one up-to-date copy
    • Write miss registers at “directory” registry
    • Shared LLC data arrays double as registry
      – Keep valid data or registered core id
DeNovo Coherence for Data (2 of 2)

- More complexity, performance, and energy efficient than MESI
  - DeNovoND adds structured locks [ASPLOS’13, Top Picks’14]
    - When to self-invalidate: at lock acquire
    - What data to self-invalidate: dynamically collected modified data signatures
    - Special hardware support for locks

But how to handle arbitrary synchronization?
Outline

• Motivation
• Background: DeNovo Coherence for Data
  • DeNovoSync Design
• Experiments
• Conclusions
Unstructured Synchronization

Michael-Scott non-blocking queue

void queue.enqueue(value v):

node *w := new node(v, null)
ptr t, n
loop

  t := tail
  n := t->next
  if t == tail
    if n == null
      if (CAS(&t->next, n, w)) break;
    else CAS(&tail, t, n)
  else CAS(&tail, t, w)

• Data accesses ordered by synchronization
  – Self-invalidate at synch using static regions or dynamic signatures
• But what about synchronization?
DeNovoSync: Software Requirements

• Software requirement: **Data-race-free**
  – Distinguish synchronization vs. data accesses to hardware
  – Obeyed by C++, C, Java, ...

• Semantics: Sequential consistency

• Optional software information for data consistency performance
DeNovoSync0 Protocol

• Key: Synch read should not return stale data

• When to self-invalidate synch location?
  – Every synch read?
  – Every synch read to non-registered state

• DeNovoSync0 registers (serializes) synch reads
  – Successive reads hit
  – Updates propagate to readers
Key: Synch read should not return stale data

When to self-invalidate synch location?
- Every synch read?
- Every synch read to non-registered state

DeNovoSync0 registers (serializes) synch reads
- Successive reads hit
- Updates propagate to readers
- BUT many registration transfers for Read-Read races
DeNovoSync = DeNovoSync0 + Hardware Backoff

- **Hardware backoff** to reduce Read-Read races
  - Remote synch read requests = hint for contention
  - Delay next (local) synch read miss for backoff cycles

- **Two-level adaptive counters** for backoff cycles
  - \( B \) = Per-core backoff counter
    - \( B \leftarrow B + I \)
    - \( I \) = Per-core increment counter
  - \( D \) = Default increment value
  - \( I \leftarrow I + D \) on \( N \)th remote synch read request
    - \( N \) determined by system configuration

More Read-Read races \( \Rightarrow \) More contention \( \Rightarrow \) Backoff longer!
Example

Test&Test&Set Lock

Thread 1   Thread 2   Thread 3

ACQ
LD
LD
LD
ACQ
LD
LD
LD
REL
ST

Backoff

Core 1
0
Registered

Core 2
0
Invalid

Core 3
0
Invalid

Shared Cache
C1
Example

Test&Test&Set Lock

Thread 1  Thread 2  Thread 3

ACQ  LD  LD  LD  ACQ  ACQ

REL  ST

BACKOFF

Shared Cache

Core 1

Backoff

Valid

Registered

16

Core 2

0

Registered

Core 3

0

Invalid

C2
Example

Thread 1
- ACQ
- LD
- LD
- CAS
- LD

Thread 2
- ACQ
- LD
- LD

Thread 3
- REL
- ST

Core 1
- Backoff: 16
- Valid

Core 2
- Invalid

Core 3
- Registered

Shared Cache
- C3

Registered: 16
Valid
Example

Thread 1    Thread 2    Thread 3

ACQ
LD
LD
LD
LD
CAS
ST
REL

Core 1
Backoff
16
Valid
HIT
Registered

Core 2
0
Registered

Core 3
0
Invalid

Shared Cache

C2
Hardware backoff reduces cache misses from Read-Read races
Outline

• Motivation

• Background: DeNovo Coherence for Data

• DeNovoSync Design

• Experiments
  – Methodology
  – Qualitative Analysis
  – Results

• Conclusions
Methodology

- Compared MESI vs. DeNovoSync0 vs. DeNovoSync
- Simics full-system simulator
  - GEMS and Garnet for memory and network simulation
- 16 and 64 cores (in-order)
- Metrics: Execution time, network traffic
- Workloads
  - 24 synchronization kernels
    - Lock-based: Test&Test&Set and array locks
    - Non-blocking data structures
    - Barriers: centralized and tree barriers, balanced and unbalanced
  - 13 application benchmarks
    - From SPLASH-2 and PARSEC 3.1
- Annotated data sharing statically (choice orthogonal to this paper)
Qualitative Analysis

• Analyze costs (execution time, traffic) in two parts
  – Linearization point
    • Ordering of linearization instruction = ordering of method
    • Usually on critical path
  – Pre-linearization points
    • Non-linearization instructions (do not determine ordering)
    • Usually checks, not on critical path
Multiple readers, one succeeds: Test&Test&Set locks

<table>
<thead>
<tr>
<th>Linearization</th>
<th>Pre-linearization</th>
</tr>
</thead>
<tbody>
<tr>
<td>MESI</td>
<td></td>
</tr>
<tr>
<td>DeNovoSync0</td>
<td></td>
</tr>
<tr>
<td>DeNovoSync</td>
<td></td>
</tr>
</tbody>
</table>
Multiple readers, one succeeds: Test&Test&Set locks

<table>
<thead>
<tr>
<th></th>
<th>Linearization</th>
<th>Pre-linearization</th>
</tr>
</thead>
<tbody>
<tr>
<td>MESI</td>
<td>Release has high inv overhead, on critical path</td>
<td>on critical path to next acquire</td>
</tr>
<tr>
<td>DeNovoSync0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DeNovoSync</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Multiple readers, one succeeds: Test&Test&Set locks

<table>
<thead>
<tr>
<th></th>
<th>Linearization</th>
<th>Pre-linearization</th>
</tr>
</thead>
<tbody>
<tr>
<td>MESI</td>
<td>Release has high inv overhead, on critical path to next acquire</td>
<td></td>
</tr>
<tr>
<td>DeNovoSync0</td>
<td>No inv overhead</td>
<td></td>
</tr>
<tr>
<td>DeNovoSync</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Multiple readers, one succeeds: Test&Test&Set locks

<table>
<thead>
<tr>
<th></th>
<th>Linearization</th>
<th>Pre-linearization</th>
</tr>
</thead>
<tbody>
<tr>
<td>MESI</td>
<td>Release has high inv overhead, on critical path to next acquire</td>
<td></td>
</tr>
<tr>
<td>DeNovoSync0</td>
<td>No inv overhead</td>
<td></td>
</tr>
<tr>
<td>DeNovoSync</td>
<td>No inv overhead</td>
<td></td>
</tr>
</tbody>
</table>
Multiple readers, one succeeds: Test&Test&Set locks

<table>
<thead>
<tr>
<th></th>
<th>Linearization</th>
<th>Pre-linearization</th>
</tr>
</thead>
<tbody>
<tr>
<td>MESI</td>
<td>Release has high inv overhead, on critical path to next acquire</td>
<td>Local spinning</td>
</tr>
<tr>
<td>DeNovoSync0</td>
<td>No inv overhead</td>
<td></td>
</tr>
<tr>
<td>DeNovoSync</td>
<td>No inv overhead</td>
<td></td>
</tr>
</tbody>
</table>
- **Multiple readers, one succeeds: Test&Test&Set locks**

<table>
<thead>
<tr>
<th></th>
<th>Linearization</th>
<th>Pre-linearization</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MESI</strong></td>
<td>Release has high inv overhead, on critical path to next acquire</td>
<td>Local spinning</td>
</tr>
<tr>
<td><strong>DeNovoSync0</strong></td>
<td>No inv overhead</td>
<td>Read-Read races, but not on critical path</td>
</tr>
<tr>
<td><strong>DeNovoSync</strong></td>
<td>No inv overhead</td>
<td></td>
</tr>
</tbody>
</table>
Multiple readers, one succeeds: Test&Test&Set locks

<table>
<thead>
<tr>
<th></th>
<th>Linearization</th>
<th>Pre-linearization</th>
</tr>
</thead>
<tbody>
<tr>
<td>MESI</td>
<td>Release has high inv overhead, on critical path to next acquire</td>
<td>Local spinning</td>
</tr>
<tr>
<td>DeNovoSync0</td>
<td>No inv overhead</td>
<td>Read-Read races, but not on critical path</td>
</tr>
<tr>
<td>DeNovoSync</td>
<td>No inv overhead</td>
<td>Backoff mitigates Read-Read races</td>
</tr>
</tbody>
</table>

DeNovo expected to be better than MESI
Similar analysis holds for non-blocking constructs
• Many readers, all succeed: Centralized barriers
  – MESI: high linearization due to invalidations
  – DeNovo: high linearization due to serialized read registrations

• One writer, one reader: Tree barriers, array locks
  – DeNovo, MESI comparable to first order

• Qualitative analysis only considers synchronization
  – Data effects: Self-invalidation, coherence granularity, …
  – Orthogonal to this work, but affect experimental results
For 44 of 48 cases, 22% lower exec time, 58% lower traffic (not shown)

Remaining 4 cases:

- Centralized unbalanced barriers: But tree barriers better for MESI too
- Heap with array locks: Need dynamic data signatures for self-invalidation
For 12 of 13 cases, 4% lower exec time, 24% lower traffic

Memory time dominated by data (vs. sync) accesses
Conclusions

- **DeNovoSync**: First to cache arbitrary synch w/o writer-initiated inv
  - Registered reads + hardware backoff

- With simplicity, performance, energy advantage
  - No transient states, no directory storage, no inv/acks, no false sharing, ...

- **DeNovoSync** vs. MESI
  - Kernels: For 44 of 48 cases, 22% lower exec time
  - Apps: For 12 of 13 cases, 4% lower exec time

⇒ Complexity-, performance-, energy-efficiency w/o s/w restrictions

- Future: DeNovo w/ heterogeneity [ISCA’15], dynamic data signatures