Speculative Synchronization: Applying Thread-Level Speculation to Parallel Applications

José F. Martínez* and Josep Torrellas
University of Illinois
ASPLOS 2002

* Now at Cornell University
Overview

- Allow speculative execution passed sync points
  - Active barriers
  - Busy locks
  - Unset flags

- Apply thread-level spec idea to explicitly parallel programs
  - Buffer speculative data (caches)
  - Detect and repair dependence violations on the fly
  - Forward progress: Keep one safe thread at all times

- ~35% sync time reduction, ~7.5% exec time reduction
Synchronization in Parallel Programs

- Synchronization to ensure correctness
  - Barriers
  - Locks
  - Flags (spinlocks)

- Applied by compilers and programmers alike
  - Parallelizing compilers: often full barriers
  - Programmers: often sync primitives
    - Macros (M4, ...)
    - Directives (OpenMP, ...)

(c) 2002 José F. Martínez - Speculative Synchronization
Problem: Conservative Synchronization

- Barries, locks, flags often placed conservatively
  - Hard-to-analyze memory access patterns
    - Pointer accesses
  - Corner cases in mostly race-free codes
    - Hashed accesses
  - Aggressive sync not affordable
    - Too time-consuming
    - Too complicated

(c) 2002 José F. Martínez - Speculative Synchronization
Technique: Thread-Level Speculation*

- Execute hard-to-analyze codes in parallel—speculatively
  - Buffer speculative data (caches)
  - Detect dependence violations
  - Roll back offending threads on the fly
  - Keep one safe thread at all times → forward progress

```plaintext
for (i=0; i<n; i++)
    a[c[i]] = ...
    a[b[i]] = ...
    a[i] = ...
```

*(a.k.a. speculative parallelization)*
Proposal: Speculative Synchronization

- Execute synchronized code concurrently—*speculatively*
- Speculate past active barriers, busy locks, unset flags
  - Buffers speculative data (caches)
  - Detect dependence violations
  - Roll back offending threads
  - Keep safe thread(s)
    - Lock: owner
    - Flag: producer
    - Barrier: lagging threads
- Mechanism: offload sync op from CPU
Important Features of our Proposal

- Unified HW support for spec barriers, locks, flags
- Concurrency possible even if conflicts
  - Forward progress guaranteed by safe thread
  - All in-order safe-to-spec conflicts tolerated
- Simple HW
  - No order among spec threads $\rightarrow$ simpler than full TLS
- No programming effort
  - Retargeted macros / directives
- Compatible with conventional sync at run time

(c) 2002 José F. Martínez - Speculative Synchronization
Example: Speculative Barrier
Example: Speculative Barrier

\[ \text{A} \quad \text{B} \quad \text{C} \]

\[ \text{BARRIER} \]

\[ \text{Safe} \quad \text{Speculative} \]
Example: Speculative Barrier

- **A** (Speculative)
- **B** (Barrier)
- **C** (Speculative)
Example: Speculative Barrier

- A
- B
- C
Example: Speculative Lock

A B C D E

ACQUIRE

Safe Speculative

RELEASE
Example: Speculative Lock

ACQUIRE

RELEASE

Safe
Speculative
Example: Speculative Lock

ACQUIRE

SAFE

SPECULATIVE

RELEASE
Example: Speculative Lock

```
A ─ B ─ C

D ─ E
```

ACQUIRE

RELEASE

Safe

Speculative
Example: Speculative Lock

ACQUIRE

RELEASE

Safe
Speculative
Speculative Synchronization Unit (SSU)

- Extends cache controller
- Simple hardware
  - 1 extra “cache line”
  - 1 Spec bit / cache line
  - Some control logic
- Modest HW overhead
  - About 2KB for
    - $L1 = 16$KB
    - $L2 = 1$MB
Speculative Lock Request

- **CPU side:**
  - Supply lock address to SSU
  - Checkpoint register file in HW

- **SSU side:**
  - Initiate T&T&S loop on lock variable

- **CPU proceeds into sync’d code**

- **Use caches as speculative buffer**
  - Set Spec bit in lines accessed speculatively
Lock Acquire and Thread Commit

- SSU acquires lock (T&S successful)
  - Gang-clears all Spec bits
    - one-shot thread commit
  - Becomes idle
- Release issued later by processor
Release while Speculative (RWS)

- CPU issues release, but SSU still active
  - SSU intercepts release by processor
  - Mark in SSU: “release issued”

- When lock becomes available, SSU:
  - Does not perform T&S
  - Gang-clears all Spec bits
    → one-shot thread commit

- Thread commits without acquiring lock
Mem Access Conflict and Thread Squash

- Leverage coherence messages
  - Request to safe line: service normally
  - Request to spec line: squash thread
    - Gang-invalidate lines marked Spec+Dirty
      → one-shot squash
    - Gang-clear all Spec bits
    - Roll back & restart at sync point

- Safe threads never squashed → forward progress
- All in-order safe-to-spec dependences tolerated

(c) 2002 José F. Martínez - Speculative Synchronization
Speculative Buffer Overflow (Cache)

- When cache about to overflow
  - Safe thread: continue as usual
  - Spec thread: stall & wait
- Stalled spec thread does not affect other threads
- Spec thread eventually becomes safe
  - Then continue as usual
Speculative Flags and Barriers

- **Flag:** Leverage spec lock support
  - Supply address, “pass” value to SSU
  - Mark in SSU: “release issued” (as in RWS)
    - Only Test—no T&S
  - Commit when “pass” read

- **Barrier:** Leverage spec flag support
  - Producer = last thread to arrive
  - If not last one, spin on flag *speculatively*
  - Last thread toggles flag

(c) 2002 José F. Martínez - Speculative Synchronization
Example: Retargeted M4 Macros

- No programming effort
- Compatible at runtime with conventional sync

<table>
<thead>
<tr>
<th>Conventional Macros (Existing)</th>
<th>Speculative Macros (Proposed)</th>
</tr>
</thead>
</table>
| LOCK(‘{ lock($1); }’)          | SS_LOCK(‘{ \textbf{if}(!issu \texttt{lock}($1))\} 
| UNLOCK(‘{ unlock($1); }’)     | SS_UNLOCK(‘{ UNLOCK($1); }’) |
| WAIT(‘{ \texttt{while}($1 != $2); }’) | SS_WAIT(‘{ \textbf{if}(!issu \texttt{wait}($1,$2)) \} 
| BARRIER(‘{ \texttt{while}($1.1f[PID] != $1.1f[PID]); \texttt{lock}($1.1 lock) \texttt{c++; if}($1.c == NUMPROC) \{ \texttt{unlock}($1.1 lock) \} else \{ \texttt{unlock}($1.1 lock) \texttt{wait}($1.1f,$1.1f[PID]) \} }’) | SS_BARRIER(‘{ \texttt{while}($1.1f[PID] != $1.1f[PID]); \texttt{lock}($1.1 lock) \texttt{c++; if}($1.c == NUMPROC) \{ \texttt{unlock}($1.1 lock) \} else \{ \texttt{unlock}($1.1 lock) \texttt{wait}($1.1f,$1.1f[PID]) \} }’) |
TLR Vs Speculative Synchronization

- Execute critical sections speculatively
  - Leverage coherence protocol
  - Guaranteed forward progress even if conflicts
  - No programming effort

**TLR**
- Philosophy: Lock-free sync
- Focuses on spec locks
- Lock-free guarantees

**Speculative Synchronization**
- Philosophy: Thread-level spec
- Spec barriers, locks, flags
- Not lock-free* (but RWS)
  - Simpler HW

*Adaptive extension—see paper
Experimental Setup

- **Node:**
  - 1GHz 4-issue dynamic superscalar processor
  - 16KB 2-way L1, 256(64)KB 8-way L2
  - SSU

- **16(64)-node CC-NUMA**
  - MESI coherence protocol

- **Uncontended RT**
  - 2ns L1, 12ns L2
  - 95ns local, 175ns neighbor
Applications

- Mix of parallel codes
  - Compiler-parallelized (Polaris): APPLU* (SPECfp95)
  - Annotated: Bisort*, MST (Olden)
  - Hand-tuned: Ocean, Barnes (SPLASH-2)
Summary of Results

- Average sync time reduction ~ 35%
  - Promising for such simple hardware

- Execution time reduction up to ~ 15%, avg. ~ 7.5%

- Room for improvement
Execution Time Reduction

Across-the-board exec time reduction ~ 7.5%
Sync Time Reduction

Large sync reduction ~ 35%
Room for improvement

(c) 2002 José F. Martinez - Speculative Synchronization
Summary

- Speculative Synchronization very effective
  - Promising speedups
- Thread-level spec works for parallel programs, too
  - Safe thread $\rightarrow$ critical path largely unaffected
  - Spec cache overflow simply stalls
- Simple hardware
- No programming effort
- Room for improvement
  - Residual sync, false sharing, ...
Speculative Synchronization: Applying Thread-Level Speculation to Parallel Applications

José F. Martínez* and Josep Torrellas
University of Illinois
ASPLOS 2002

* Now at Cornell University