Reducing Memory Buffering Overhead in Software Thread-Level Speculation

Zhen Cao — zhen.cao@mcgill.ca

Clark Verbrugge — clump@cs.mcgill.ca

McGill

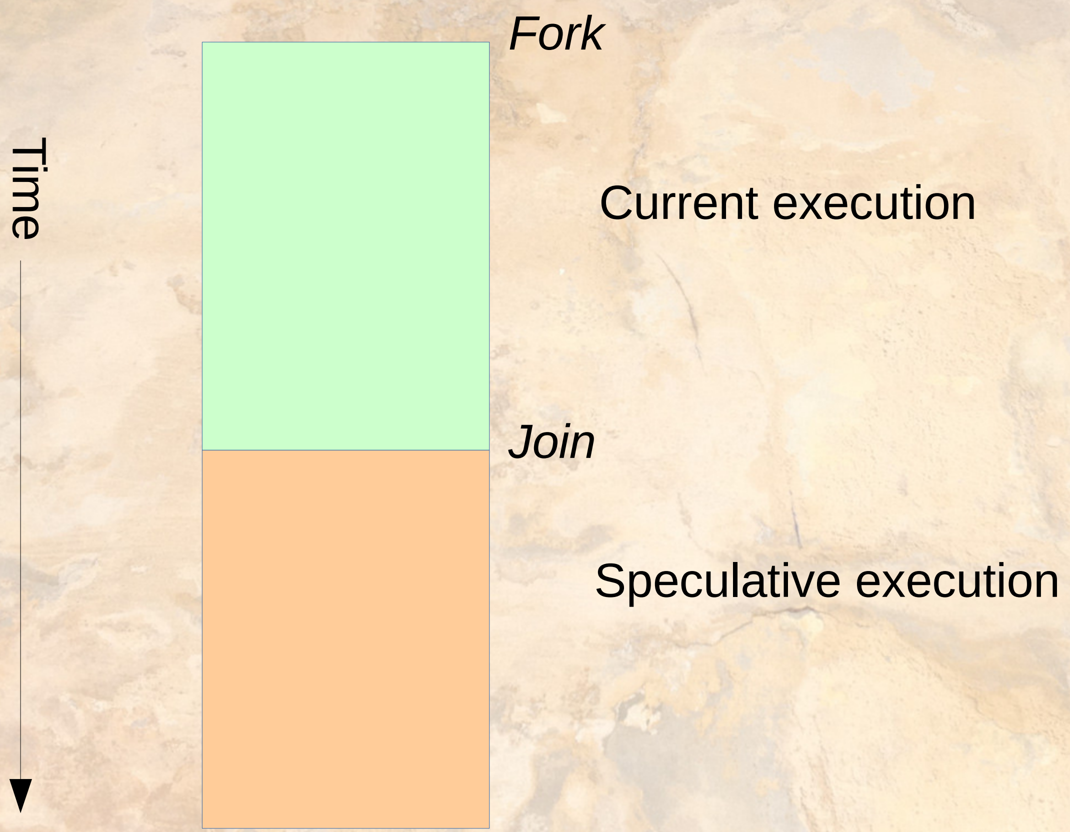
Compiler Construction 2016

Thread Level Speculation (TLS)

- Basis for automatic parallelization
  - Start from sequential program
  - Optimistically parallelize future execution
    - Safety guaranteed
- Traditionally a hardware technique
- <u>Software</u> implementation
  - Large parallel granularity, no special hardware
  - Overhead concerns

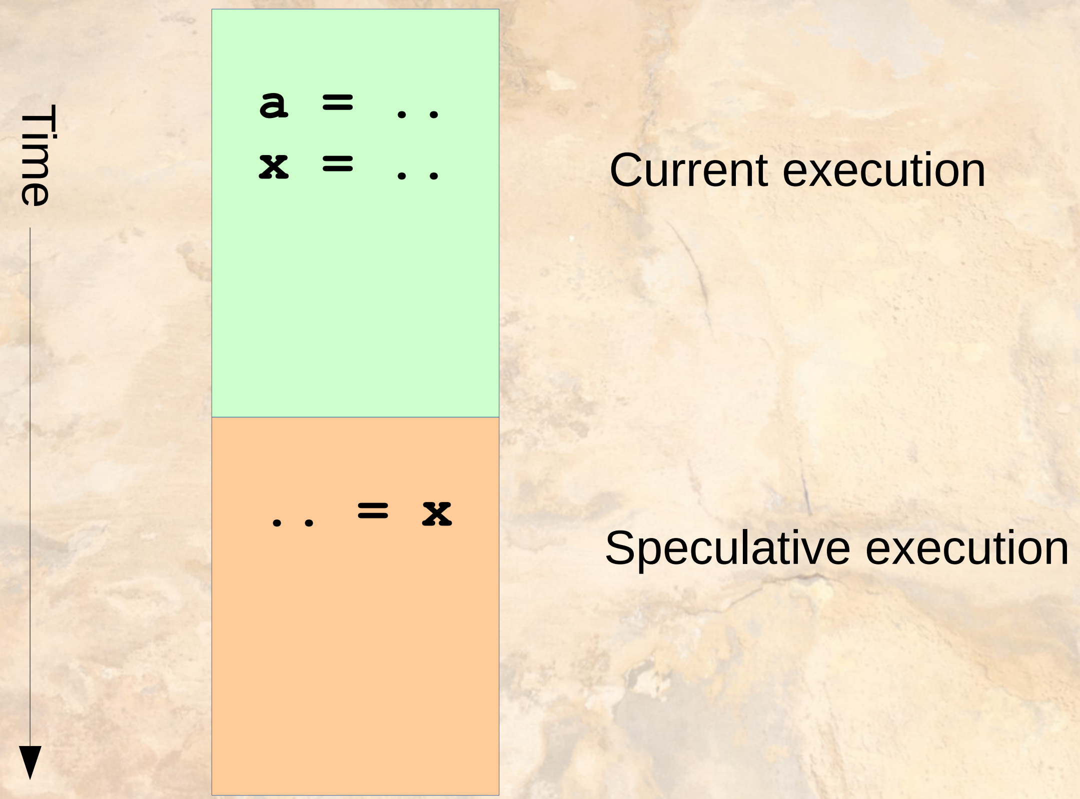
Thread Level Speculation (TLS)

- Speculative threads execute in the future



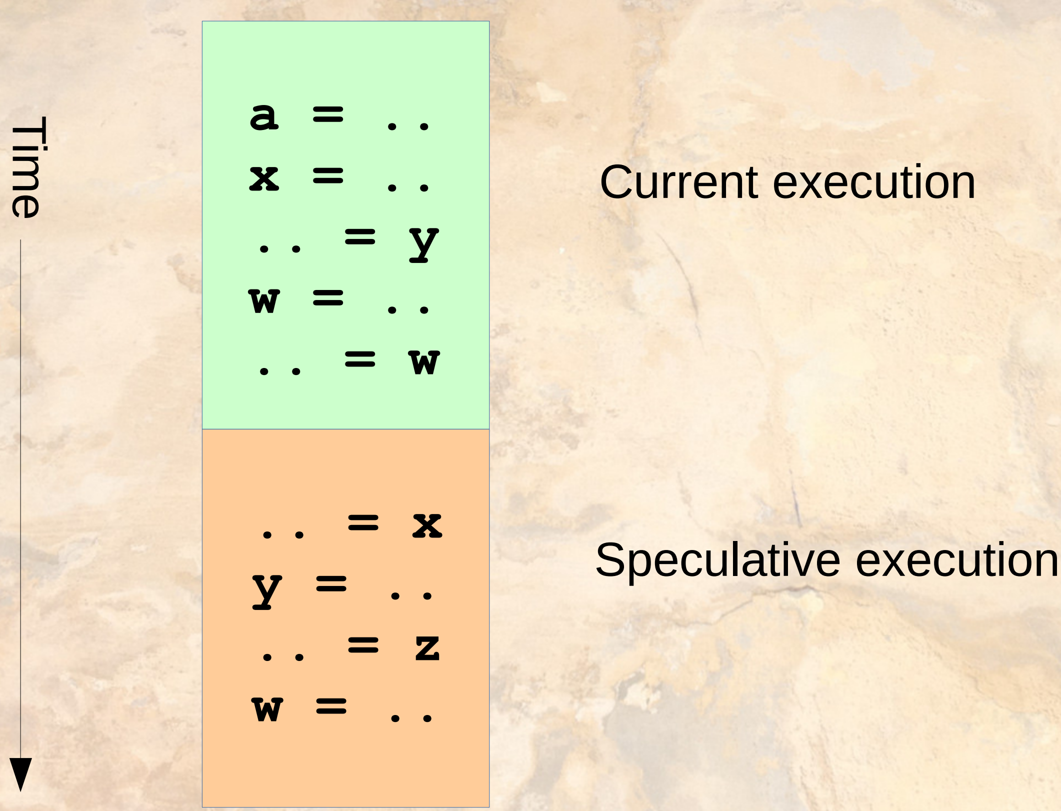
Thread Level Speculation (TLS)

- Past should affect the future



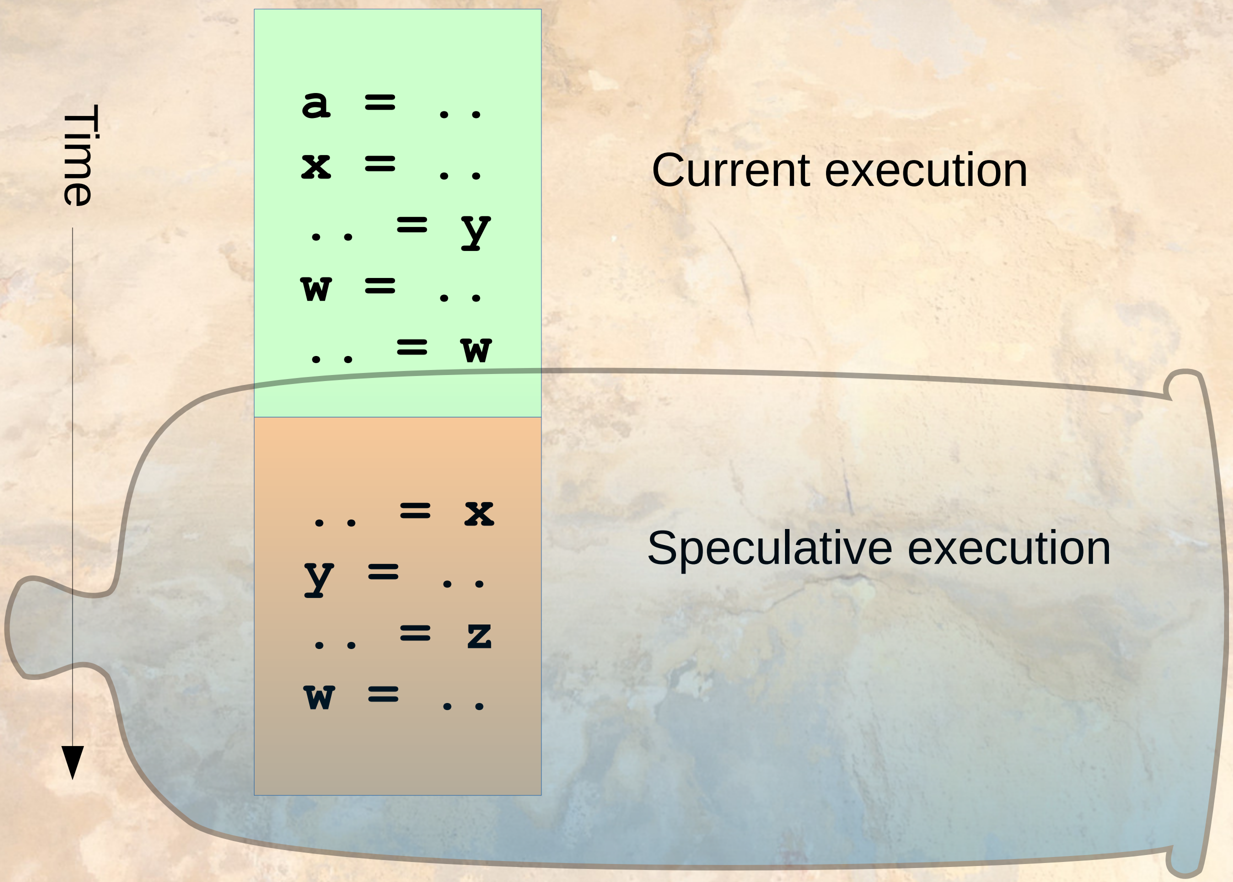
Thread Level Speculation (TLS)

- Future should not affect the past



Thread Level Speculation (TLS)

- Future should not affect the past



Contents

- Version management
  - Lazy Buffering
    - Optimized design
  - Eager Buffering
    - Optimized design
- Integrating lazy & eager
- (Thread coverage)
- Experiments

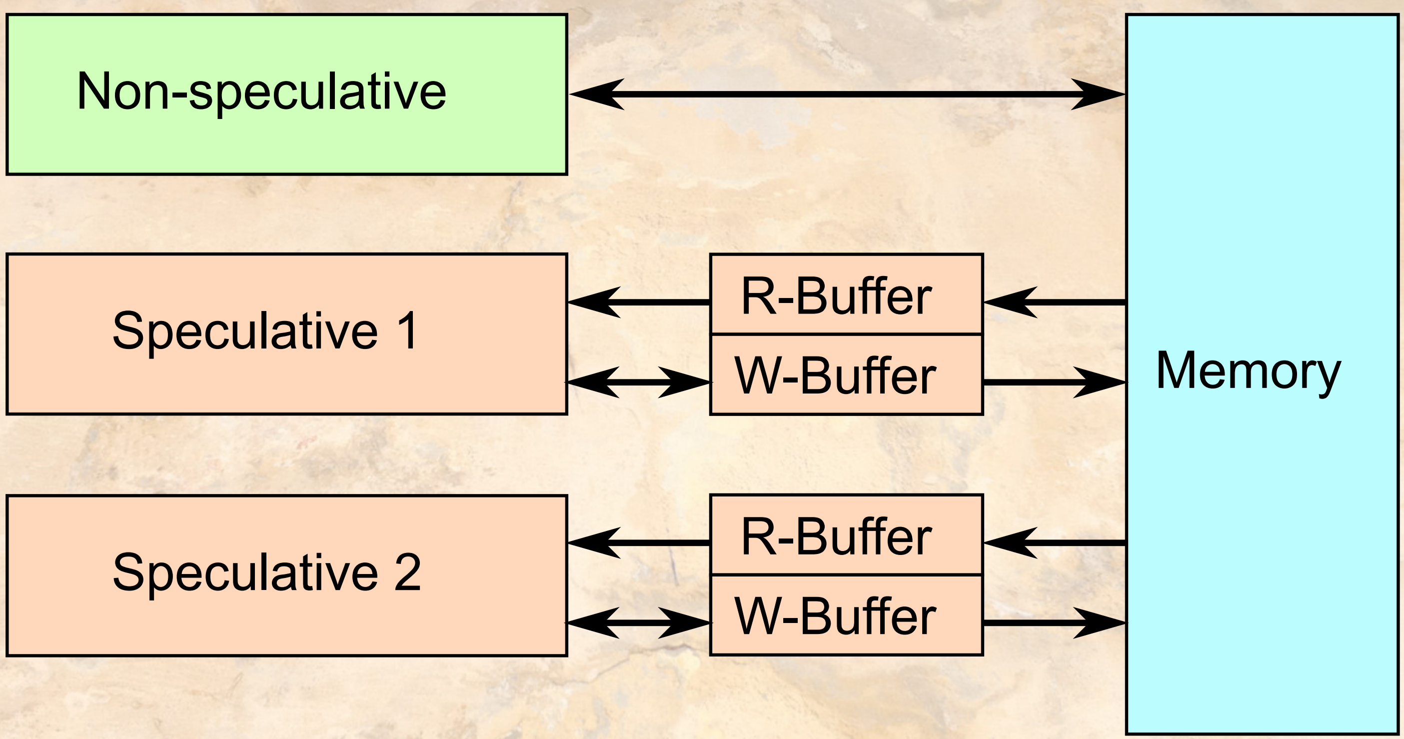
Version Management

- Key property for safety
  - Need to avoid RAW, WAR, WAW errors
  - Isolate and/or restore
- Two main flavours: Lazy & Eager

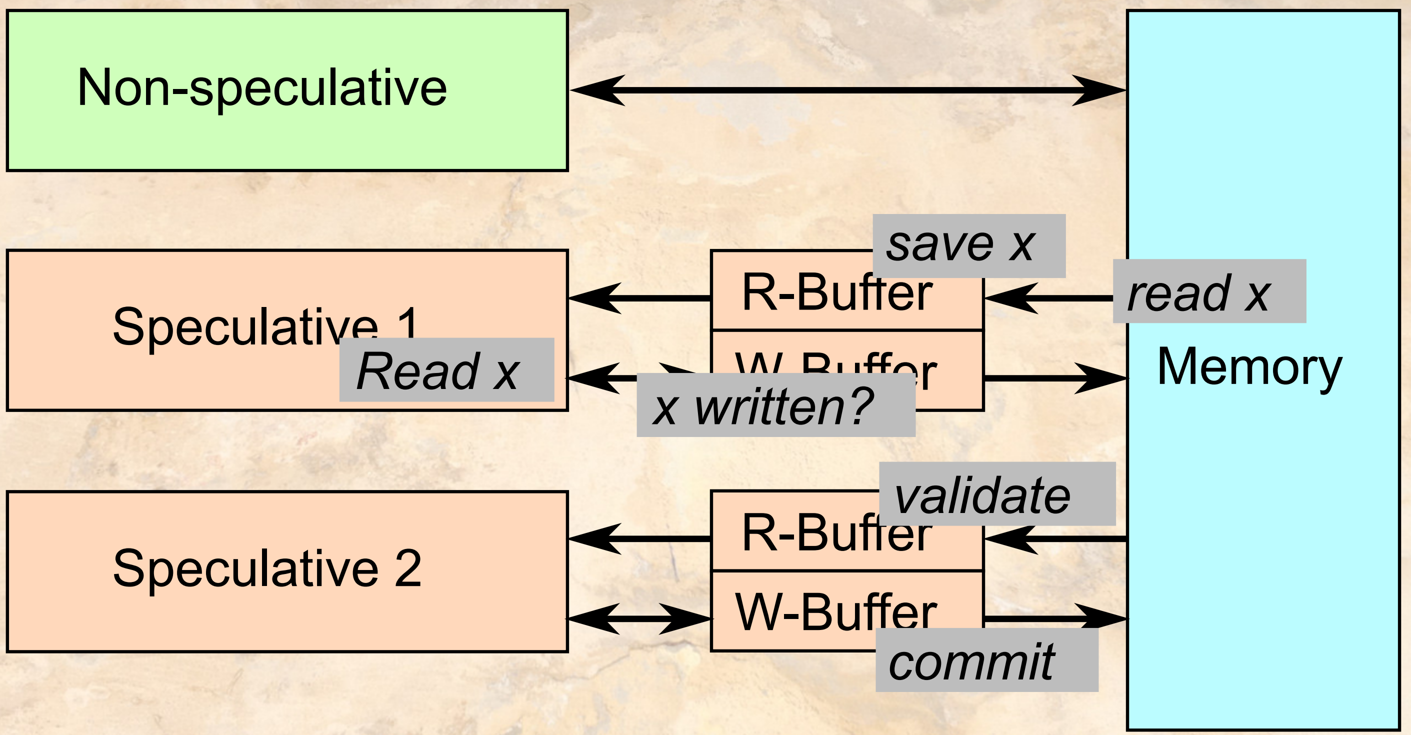
Lazy Buffering

- Lazy version management
  - Non-speculative thread accesses memory
  - Speculative threads buffered
    - Reads for validation
    - Writes for isolation

Lazy Buffering



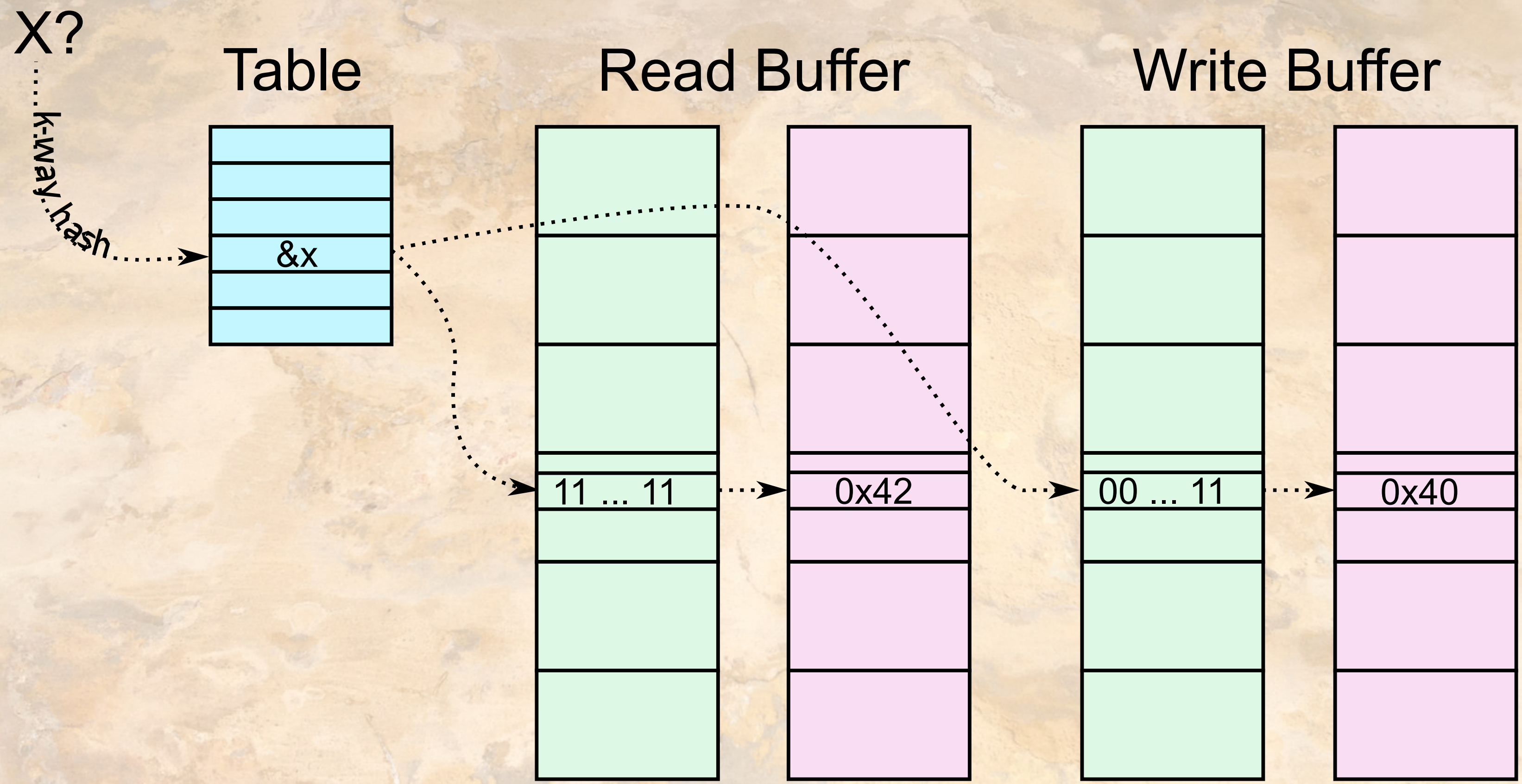
Lazy Buffering



Lazy Improvements

- Problem: idleness, due to validation/commit
  - Bigger granularity = large buffers
- Parallelize V/C?
  - Processors idle anyway...
- Coarse and fine-grain parallelization
  - But need to structure buffers to help

Per-Thread Page Tables



Per-Thread Page Tables

- Different pages committed in parallel
  - Partitioned (pages), guaranteed separate
- V/C can be vectorized on a page
  - SIMD acceleration
- Supports mixed data types

- Extra cost
  - More space, hashing

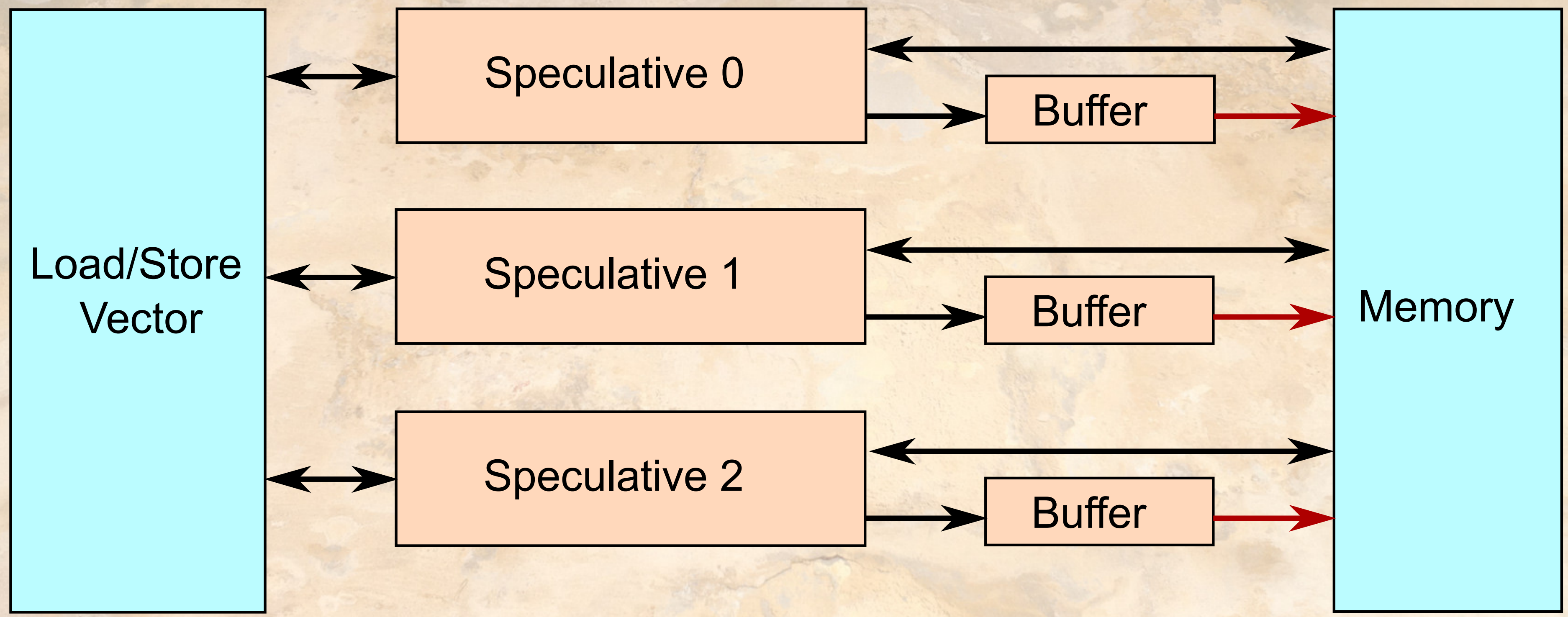
Contents

- Version management
  - Lazy Buffering
    - Optimized design
  - **Eager Buffering**
    - Optimized design
- Integrating lazy & eager
- (Thread coverage)
- Experiments

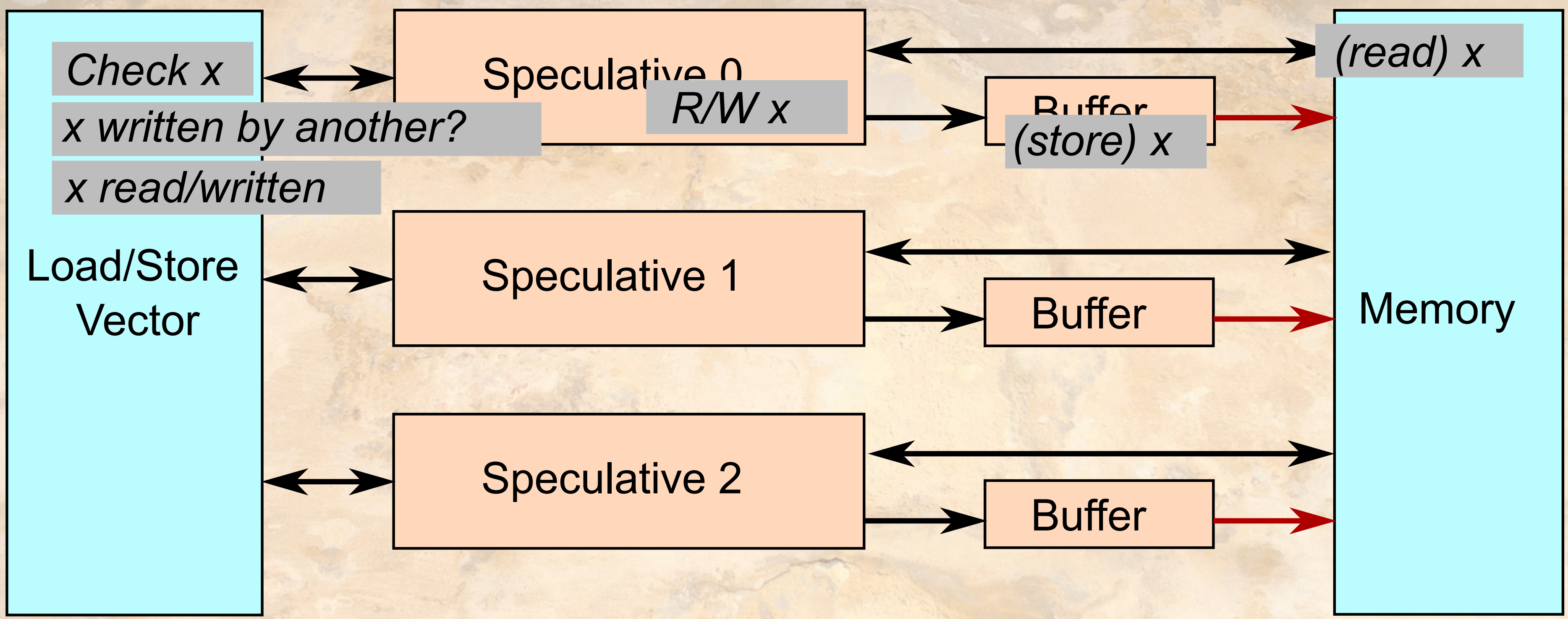
Eager Buffering

- Used in SpLIB, MiniTLS, ...
- All speculative, access main memory directly
  - Keep shadow buffer for rollback
  - Track versions for proper restore

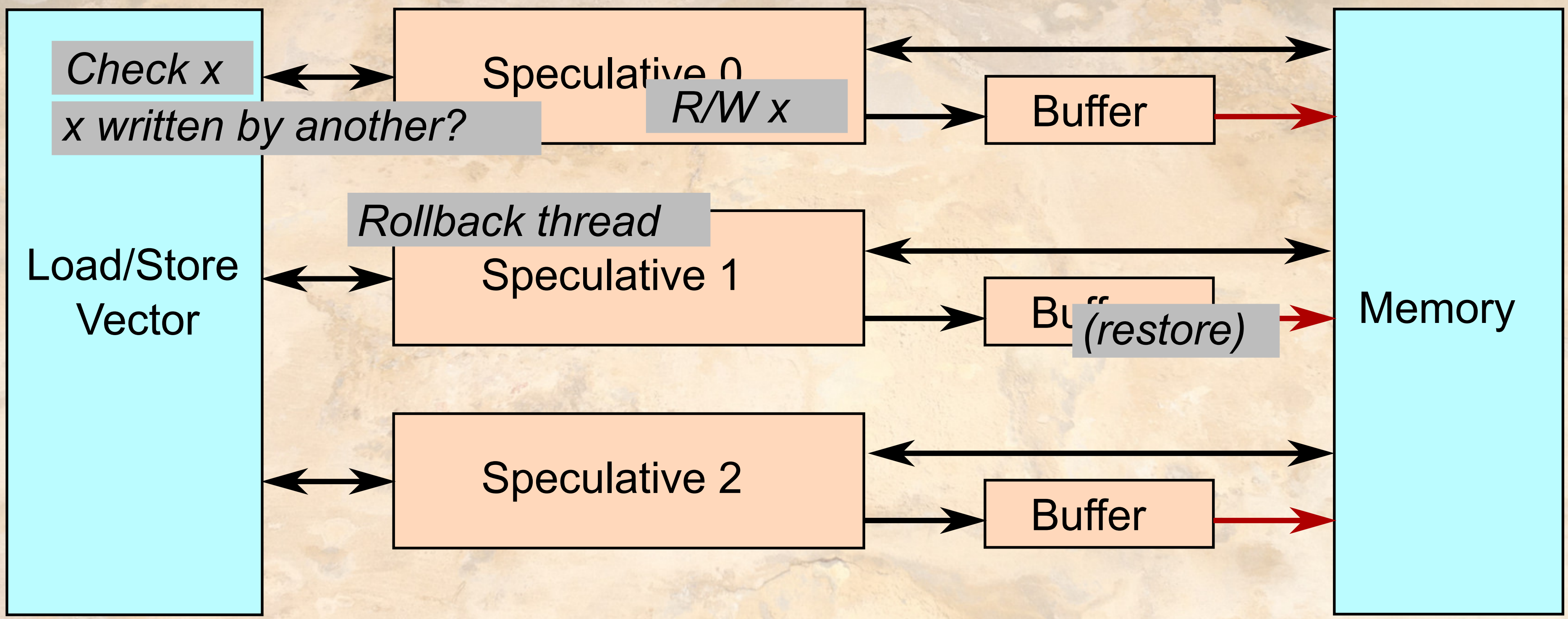
Eager Scheme



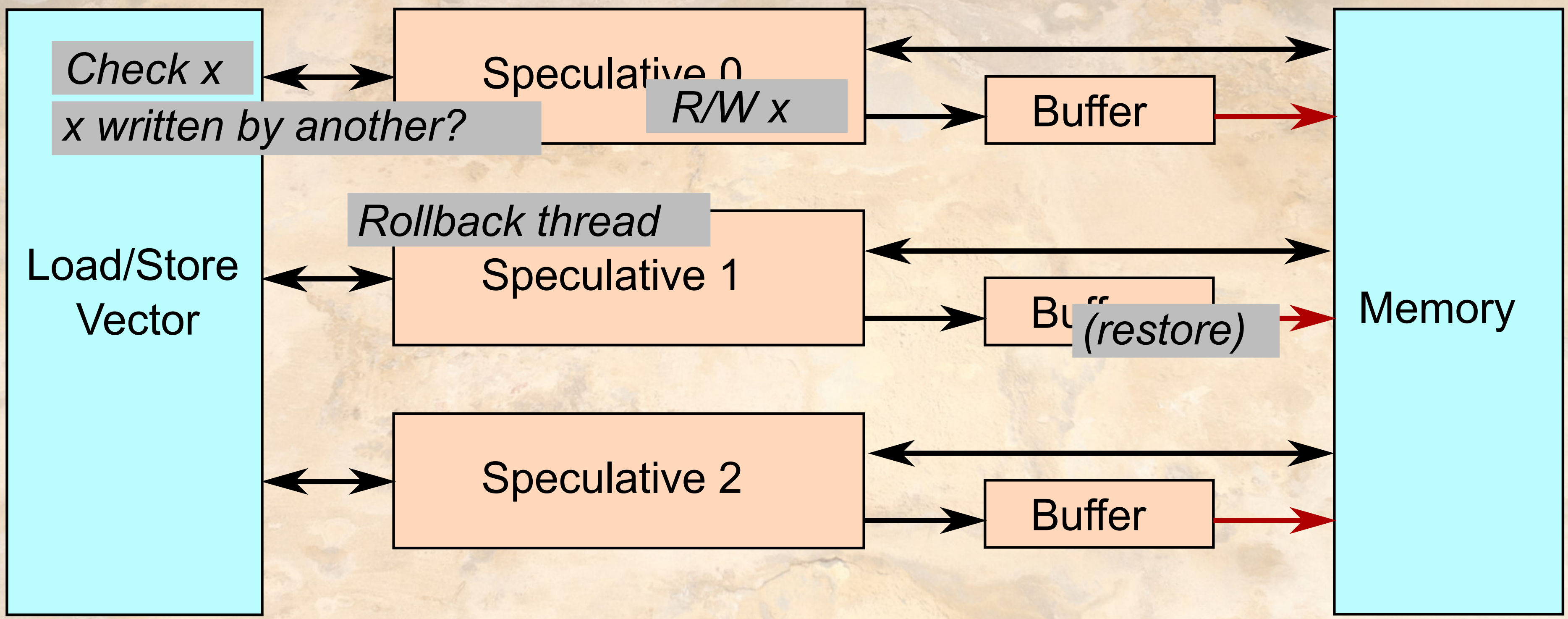
Eager Scheme



Eager Scheme



Eager Scheme



- Faster (no) commit, but slower rollback
- WAR and WAW dependencies require rollback

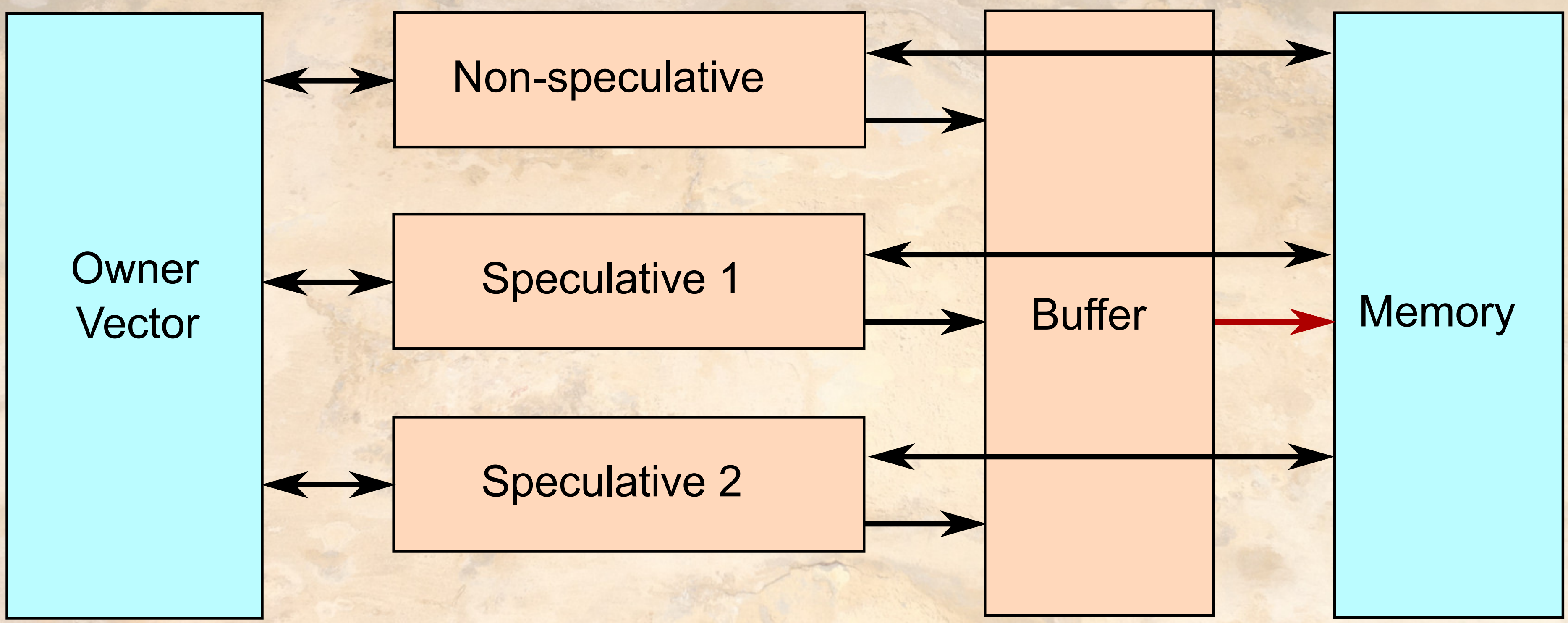
Eager Improvements

- Problem: multiple buffers, lots of versions
- Only keep one version?
  - Need to consider thread order

Shared Address-Owner Buffering

- Single shadow buffer for all threads
  - At most one buffered copy of each variable
- Improved space
  - *O(D) vs O(D+W)*
    - *D* data accesses, *W* number of writes
- Finer or coarser granularity
  - Treat vars as WORD bytes

Shared Address-Owner Buffering



Shared Address-Owner Buffering

- Each WORD has
  - Owner for dependency tracking
    - One bit for whether written or not
  - Shadow copy for rollback
- Owners ordered
  - In-order forking
    - Global counter ok
  - NS lowest

not owned

$t < t_0$	?

only read by t

$t_0 \leq t \leq t_{max}$	0

written by t

$t_0 \leq t \leq t_{max}$	1

shared

111 ... 111	0

Contents

- Version management
  - Lazy Buffering
    - Optimized design
  - Eager Buffering
    - Optimized design
- **Integrating lazy & eager**
- (Thread coverage)
- Experiments

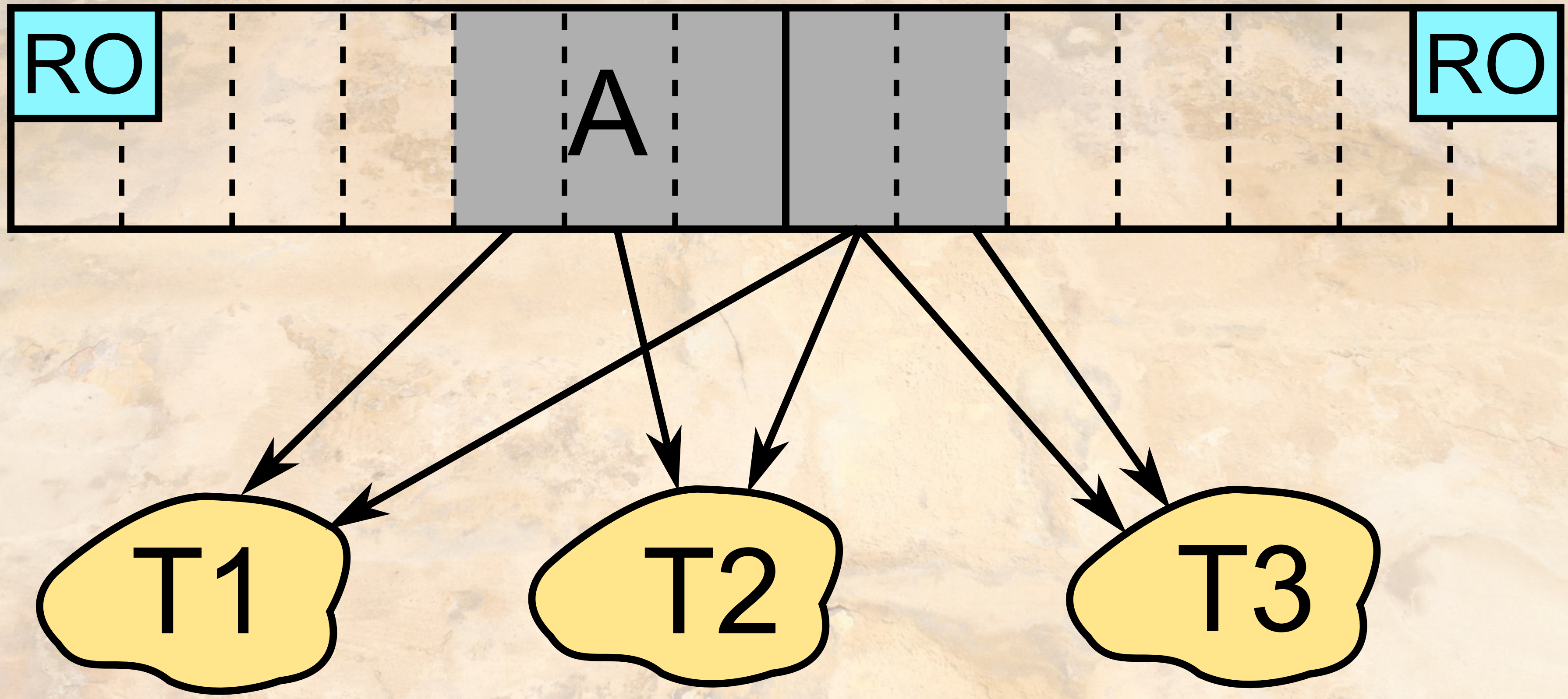
Buffering Integration

- Builds on a **readonly** optimization
- Lots of speculative regions have readonly vars
  - Fills up buffer space
- Static is rare; dynamic is not easy to identify
  - Most approaches manual, profile-based
- Just need transitively readonly
  - Within a speculative region

Readonly

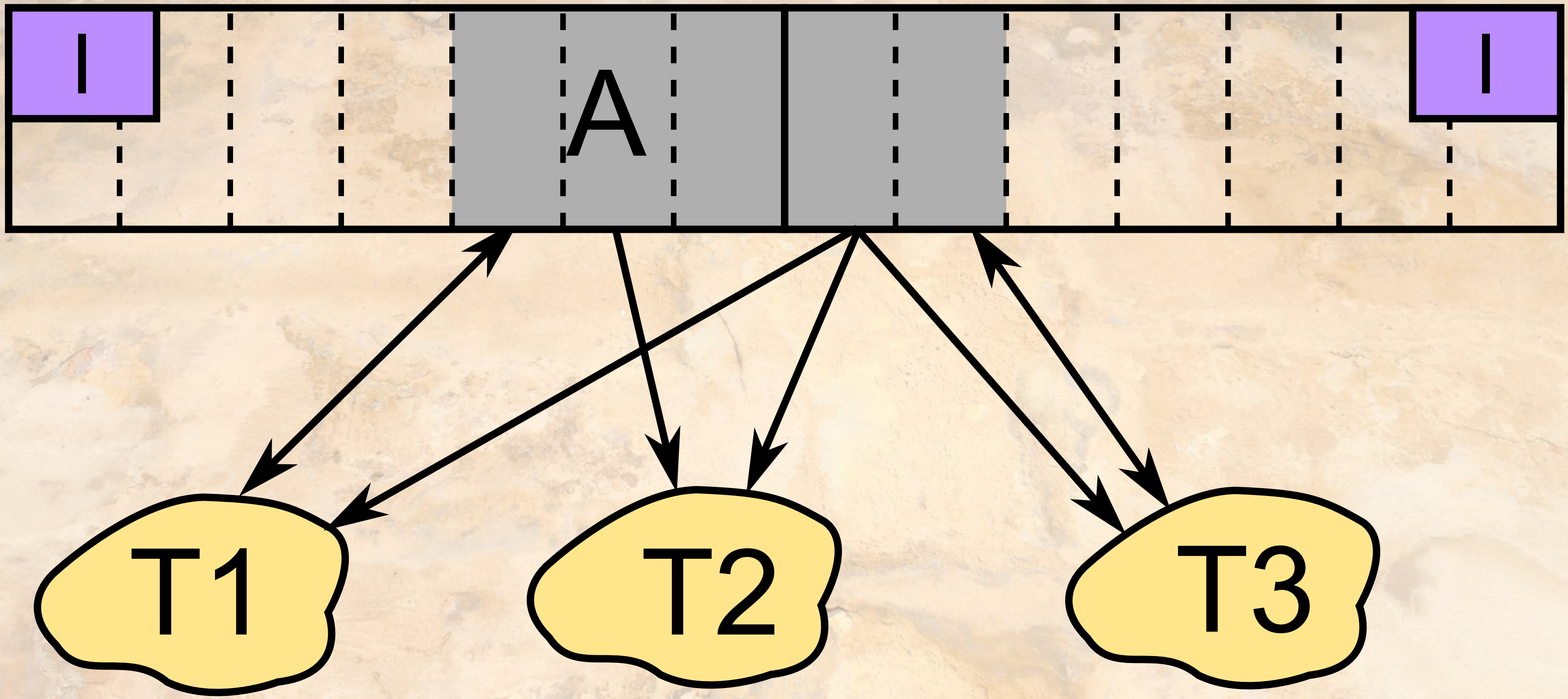
- Page based; work with larger chunks of mem
  - Heap alloc sites as single vars
- Degrees of readonly-ness
  - Readonly (default on entry)
  - Independent (threads R/W different parts)
  - Dependent (conflicting)
- Rollback reduces degrees

Readonly



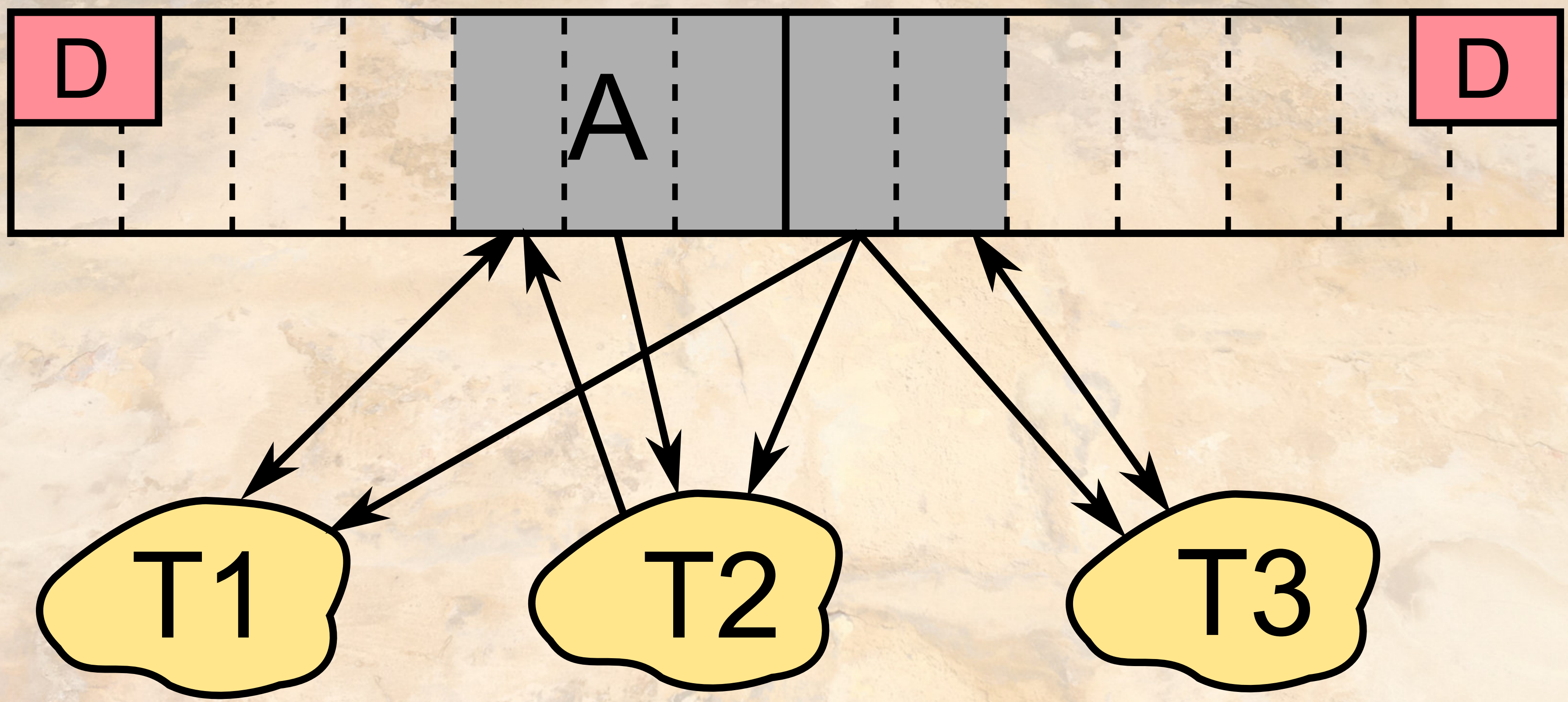
Readonly pages – no buffering required

Readonly



Independent pages – use **Eager Buffering**

Readonly



Dependent pages – use **Lazy Buffering**

Contents

- Version management
  - Lazy Buffering
    - Optimized design
  - Eager Buffering
    - Optimized design
- Integrating lazy & eager
- (Thread coverage)
- **Experiments**

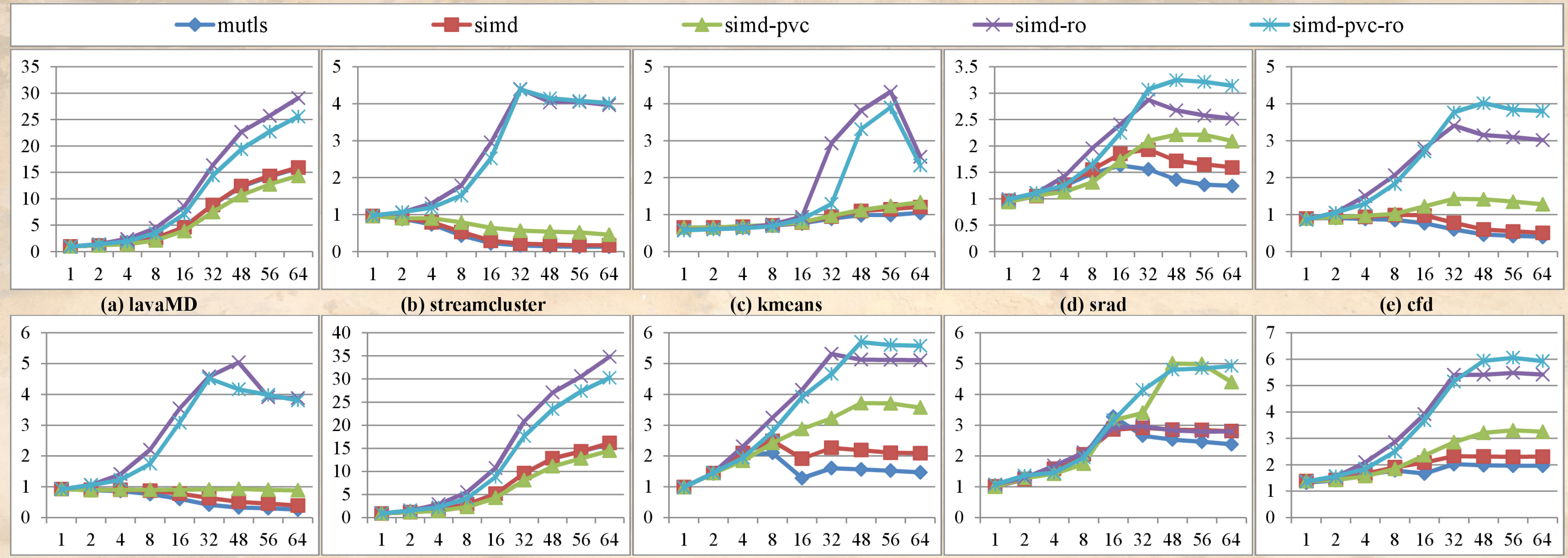
Experiments

lavaMD	C	Rodinia
streamcluster	C++	Rodinia
kmeans	C	Rodinia
srad	C	Rodinia
cfd	C++	Rodinia
sparsematmul	C	SciMark
smallpt	C++	smallpt
bwaves	Fortran	SPEC CPU2006
fft	C	MUTLS

AMD Opteron 6274 (4x16 cores, 64GB memory)

- Parallel V/C cores: 0-7

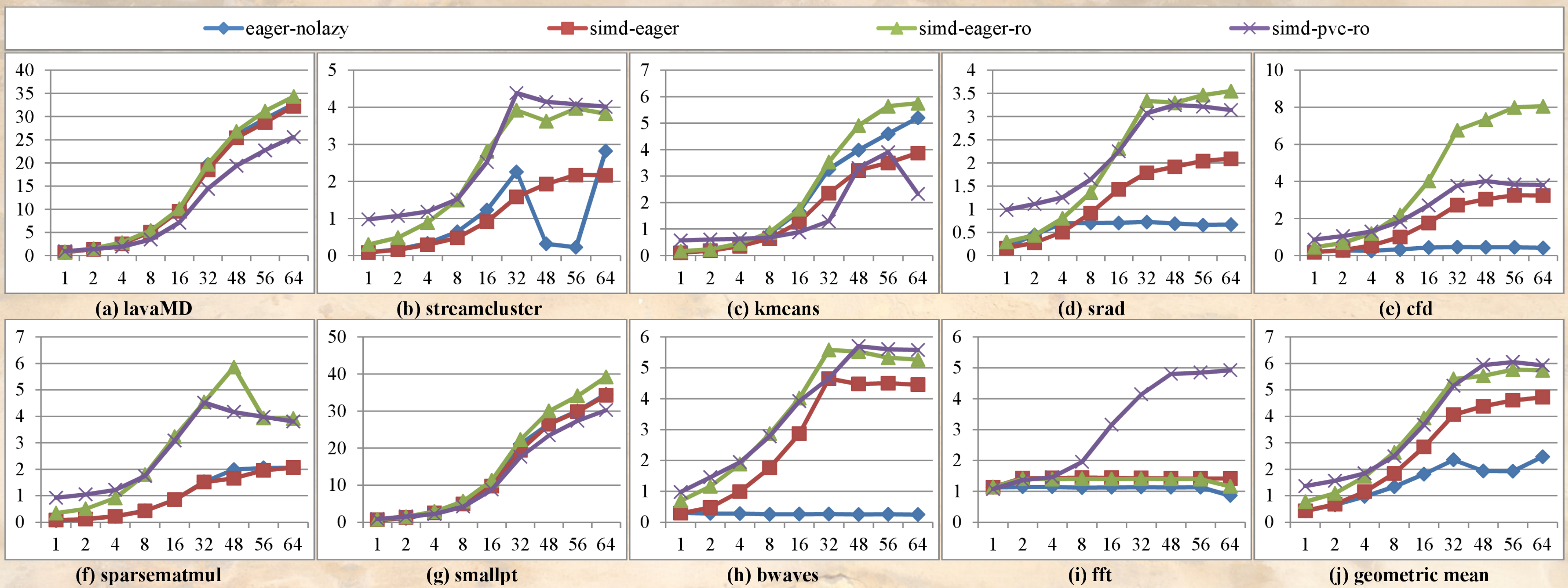
Experiments



(a) lavaMD (b) streamcluster (c) kmeans (d) srad (e) cfd
(f) sparsematmul (g) smallpt (h) bwaves (i) fft (j) geometric mean

mutls	Plain lazy
simd	Lazy with SIMD V/C
simd-pvc	Add parallel V/C
simd-ro	Lazy with SIMD and Readonly
simd-pvc-ro	Add parallel V/C

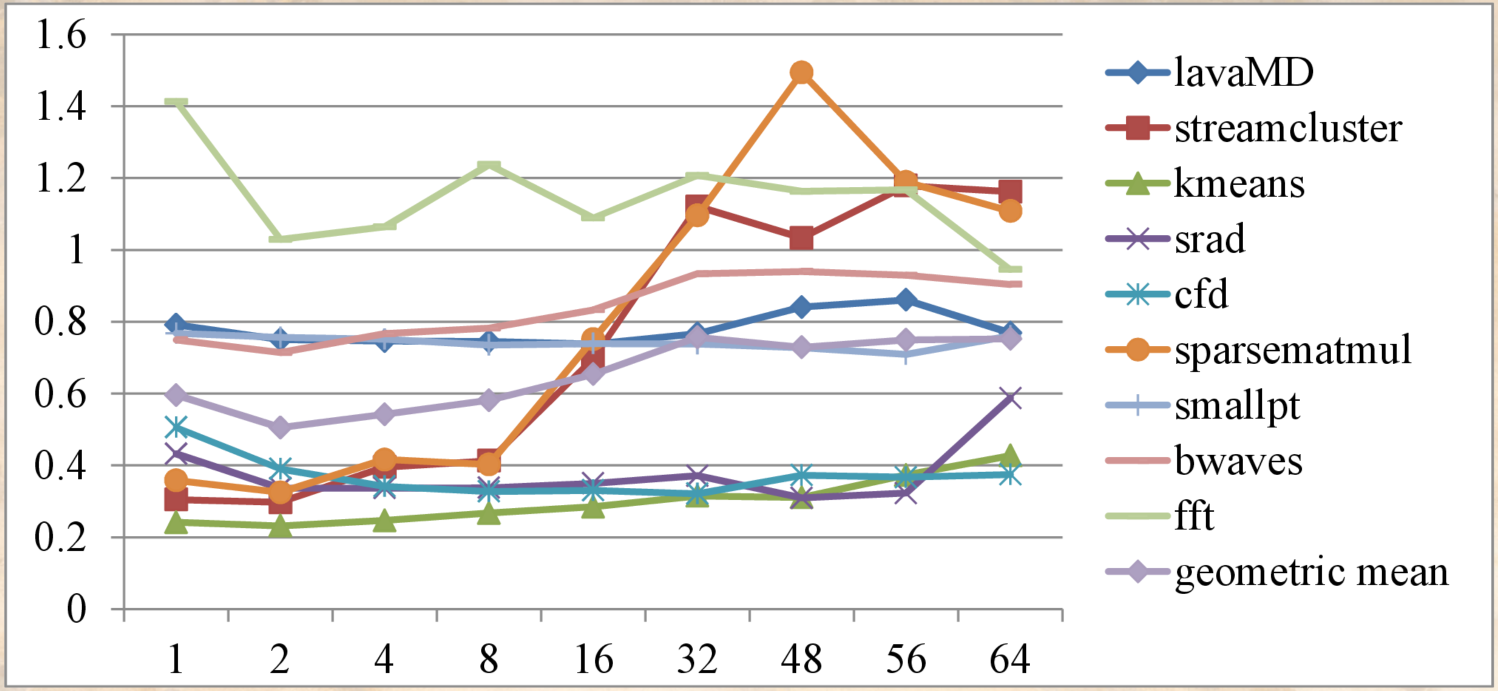
Experiments



eager-nolazy	Eager only
simd-eager	Eager, fallback to lazy SIMD V/C for dependent
simd-eager-ro	Readonly as well
simd-pvc-ro	As previous slide

Experiments

Nb: Scaled to OpenMP (manual)



Conclusions & Future Work

- Software TLS
  - Feasible, but a significant engineering effort
    - Different benchmarks need different optimizations
- Adaptivity
  - Effective, better tuning would help
- Hardware (transactional) help?
  - Small buffers!
  - Hard to enforce sequential commit

Thank You

Questions
?

Zhen Cao
zhen.cao@mail.mcgill.ca
http://www.sable.mcgill.ca/~zcao7/mutls/

Clark Verbrugge
clump@cs.mcgill.ca

Extra Slides

Adaptive Selection

- Buffering integration defaults to eager
- But costly to non-speculative thread
  - Small V/C, lazy is faster
- Adaptive heuristics, based on profiling
  - Start with lazy (more robust to rollback)

Adaptive Selection

- Compute at commit:
  - m: # memory accesses
  - T_W: work time
  - T_V: validation/commit time
  - C: overhead on var access by speculative thread (20)
  - K: delay on thread for eager case (8)
- Estimate (future) lazy:eager time over L iters:
  - Lazy speedup $S = 1+(n-1)*(T_W-C*m)/T_W$
  - $Lazy = L*T_W/S+L*T_V$
  - $Eager = L*T_W*K/n$

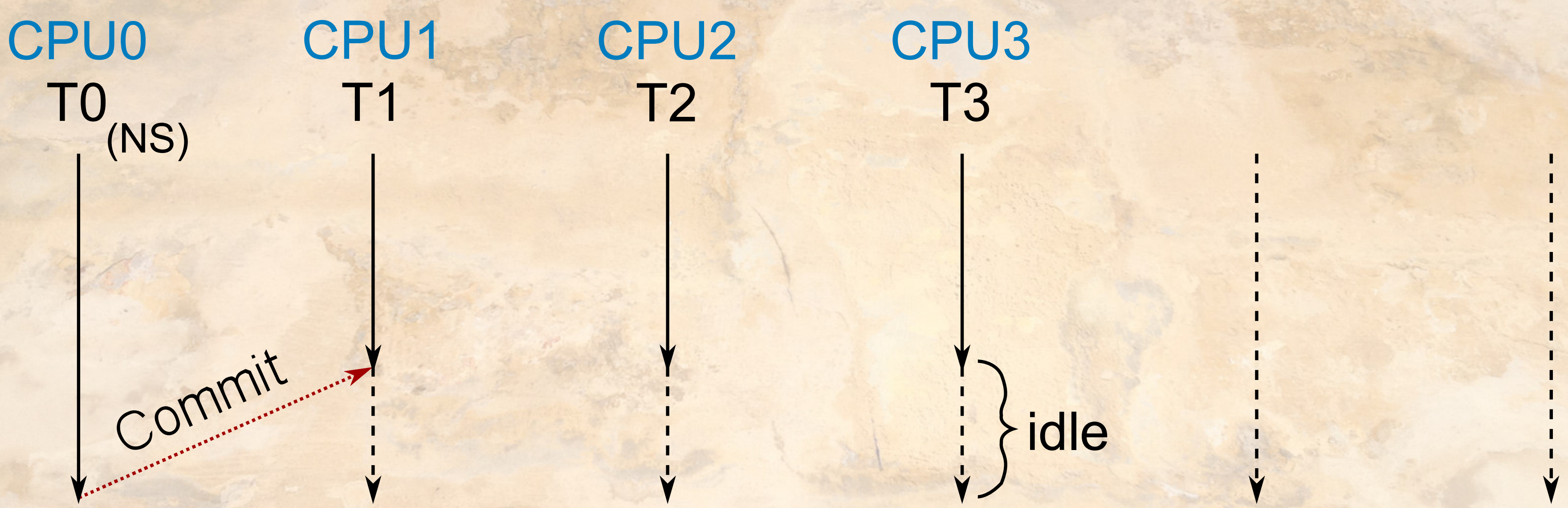
Contents

- Version management
  - Lazy Buffering
    - Optimized design
  - Eager Buffering
    - Optimized design
- Integrating lazy & eager
- **Thread coverage**
- Experiments

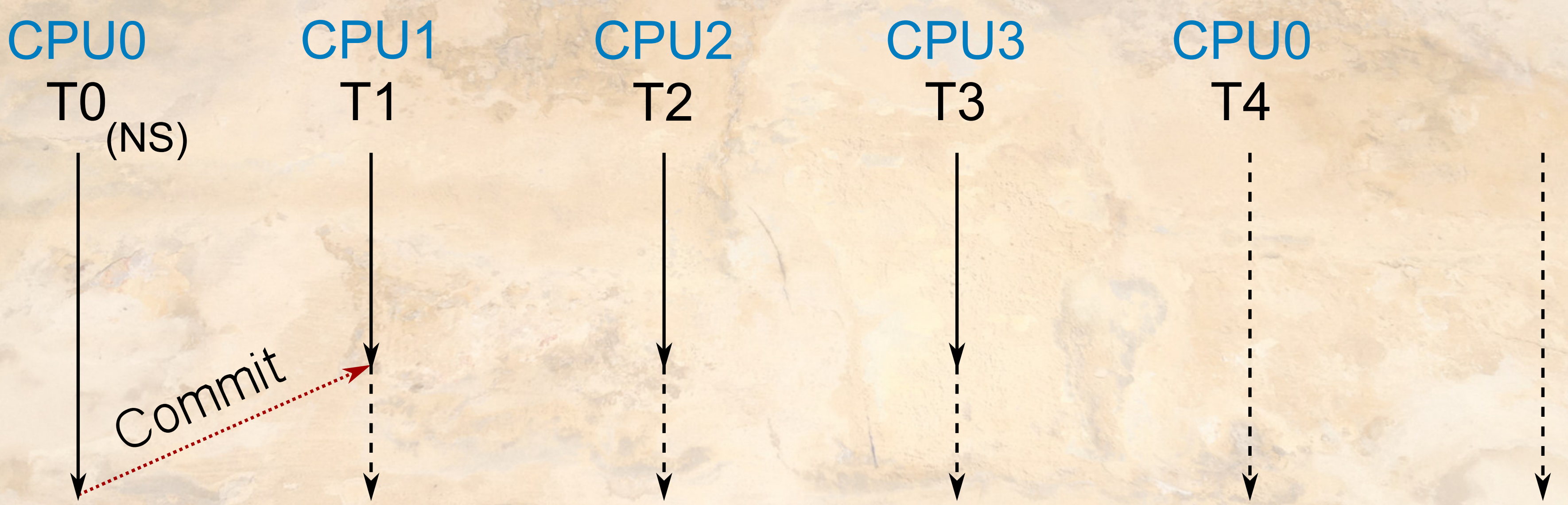
Thread Coverage

- Usually #threads ≤ #CPUs
- Speculative threads are slower
  - Waiting to join wastes resources
- Generate more speculative tasks than CPUs
  - For in-order forking, just 1 is sufficient
  - (Generally at most one pending task per thread)

Thread Coverage



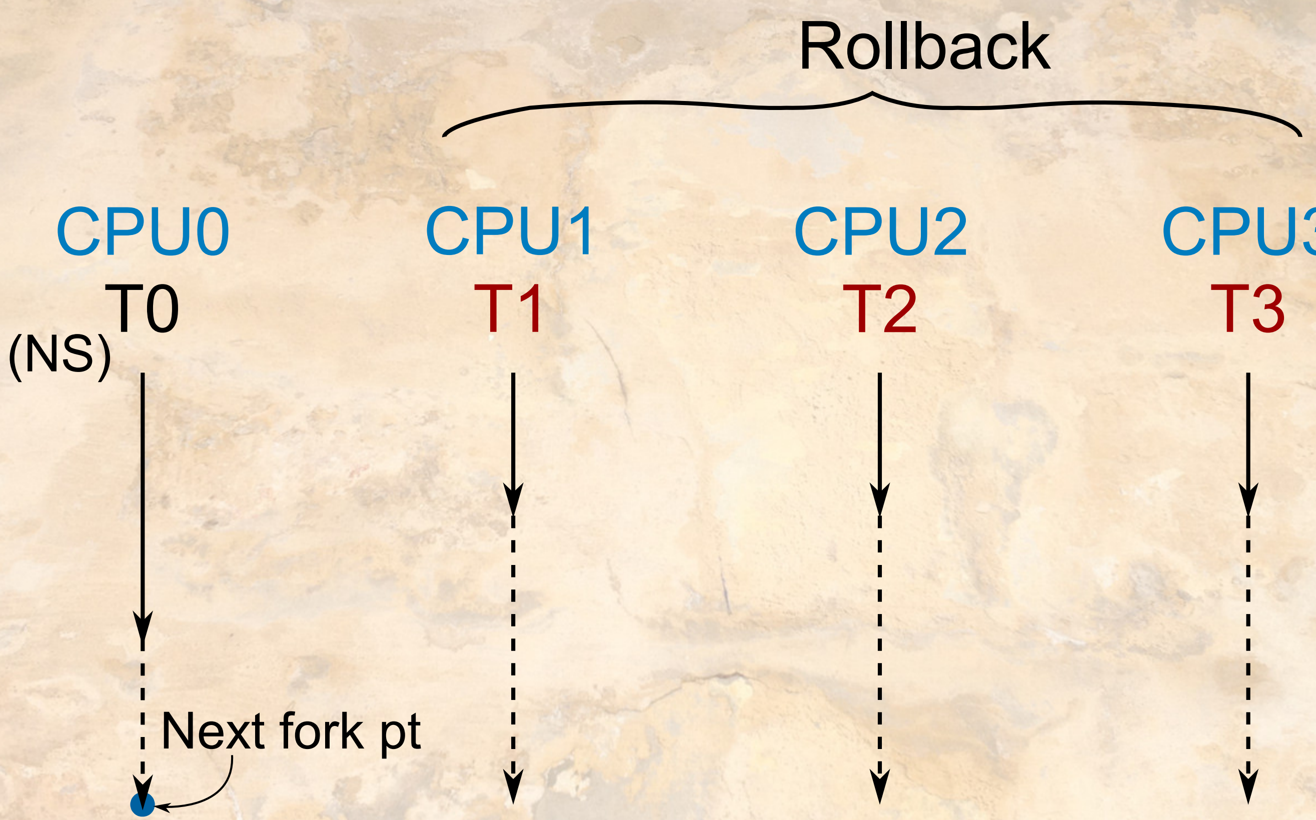
Thread Coverage



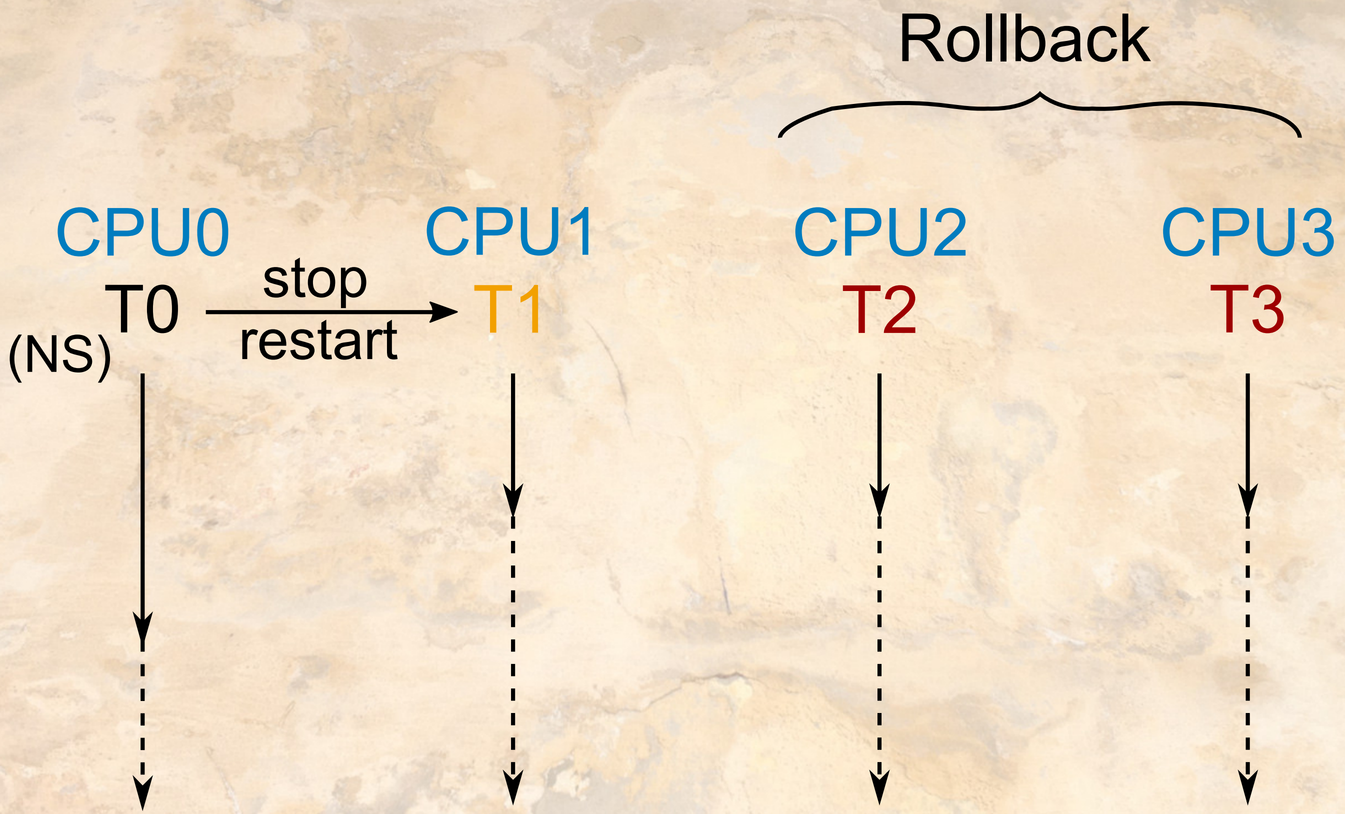
Thread Stopping

- Rollback in integration can reduce coverage
  - Rollback all speculative threads
  - Speculation continued when next fork point reached
    - No speculation until then
- Stop direct child(ren)
  - Rollback indirect children
- Reset buffering
  - Restart stopped child(ren)

Thread Stopping

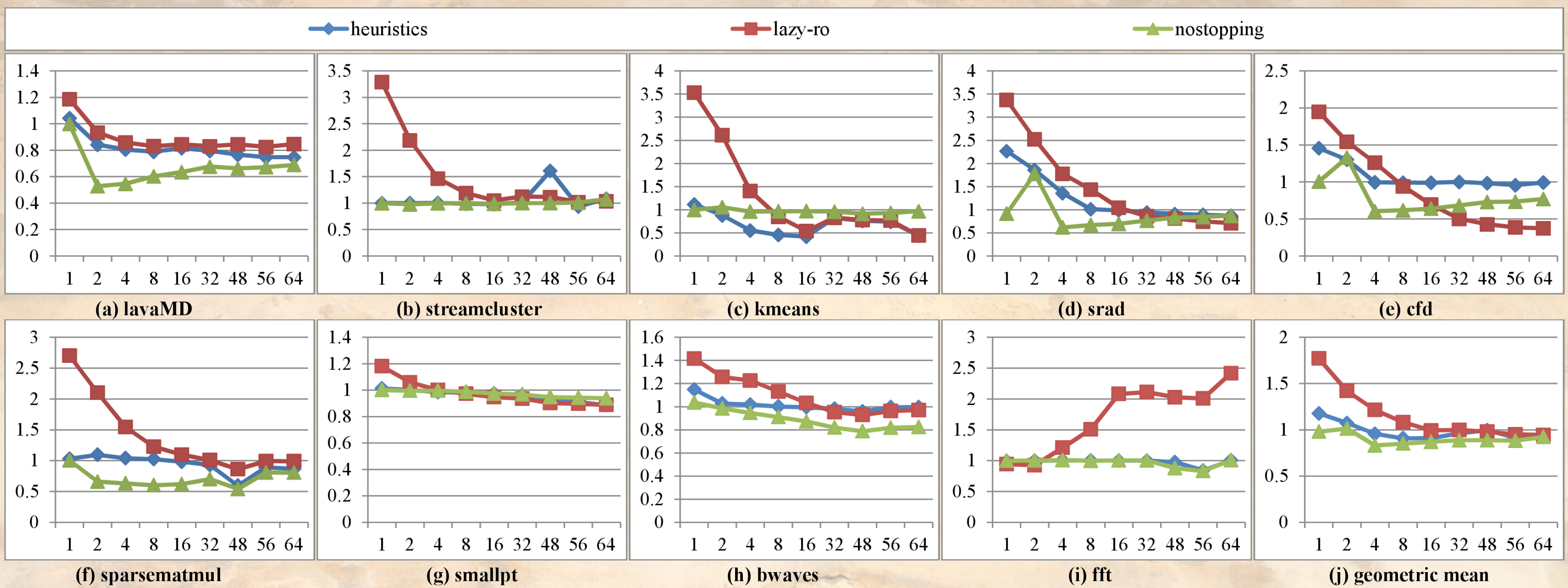


Thread Stopping



Experiments

Nb: Scaled to simd-eager-ro



heuristics	Adaptive buffering heuristics (on simd-eager-ro)
lazy-ro	Compare heuristics to lazy version
nostopping	Disable nostopping (on simd-eager-ro)