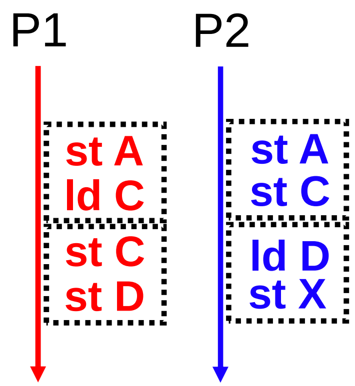
FlexBulk: Intelligently Forming Atomic Blocks in Blocked-Execution Multiprocessors to Minimize Squashes

Rishi Agarwal and Josep Torrellas

University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

Blocked-Execution Multiprocessors

- Processors continuously execute/commit chunks of instructions at a time
- Chunks execute atomically (using state buffering)

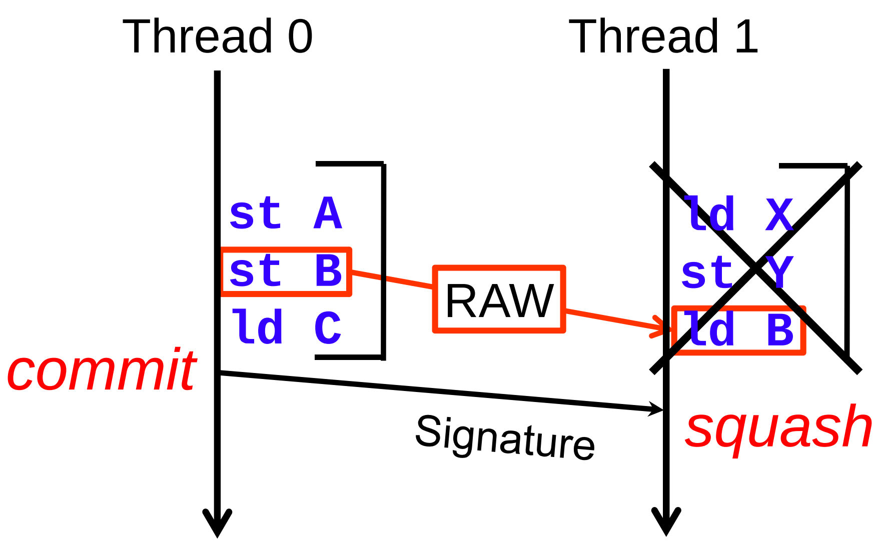


[Bulk: CommACM 09]

- Chunks can be HW-driven or generated by the compiler
- Examples: TCC, Bulk, InvisiFence, etc
- Advantages:
  - Boost performance and software productivity [TM, DetReplay]
  - Enable unique (unsafe) compiler opts [Neelakantam, BulkCompiler]

Squashes: Discarding In-Progress Chunks

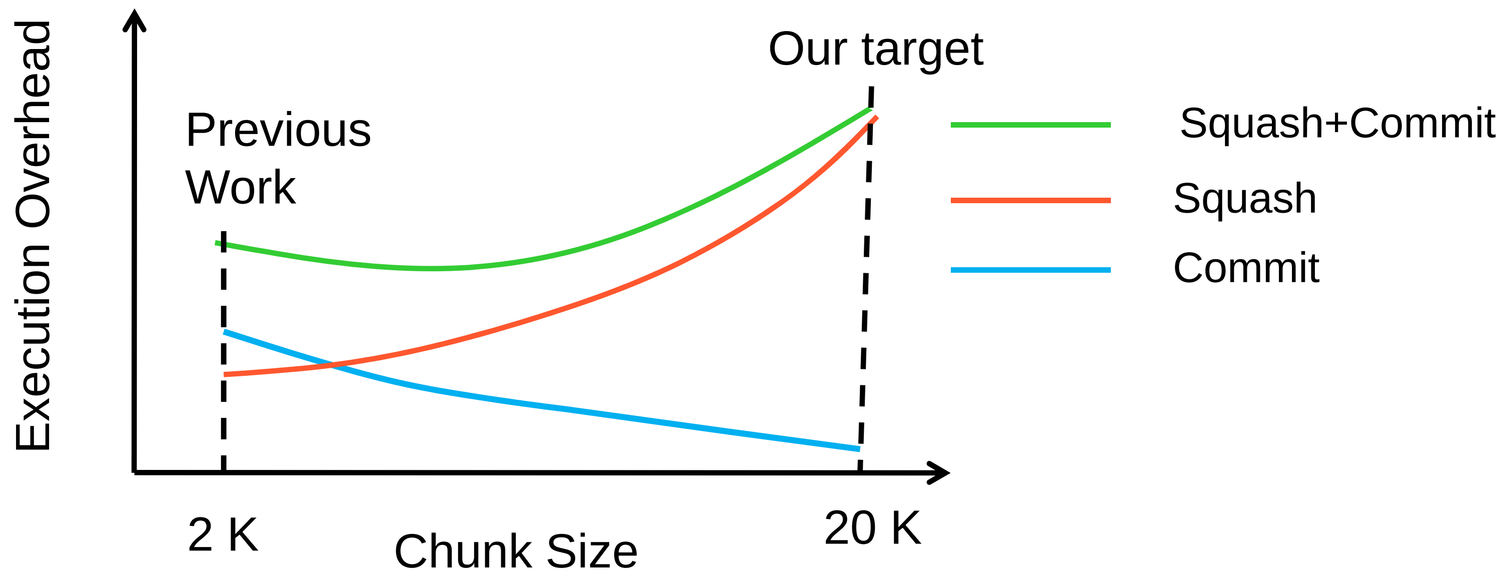
- While chunk is executing: no other proc. can observe intermediate state
- We use a Lazy approach: at end of chunk, check for data overlap (conflicts)
  - Send Signature of addresses accessed



- Chunk commit is expensive in Lazy schemes
- Other reasons that lead to squashes: False sharing and cache overflow

Compiler-Driven Chunks: Chunk Size Trade-offs

- Bigger chunk size
  - \+ More opportunities for compiler optimization
  - \+ Amortize commit cost
  - \- More chances of squashes



Contributions

- FlexBulk: Intelligently form large chunks while minimizing squashes
  - A (mostly) software framework that profiles & transforms the code (could be part of a dynamic compiler)
  - Characterize Squash Hazards: operations that frequently cause squashes
  - Tailor chunks to minimize squashes
- Results for 16-processor runs
  - Eliminate 90% of the squash time for 17K-instruction chunks
  - Avg speedup of 1.43x compared to 2K-instruction chunks

Big Picture: Compiler-Driven Chunk Generation

- Approach:
  - Insert chunk boundaries at strategic points
  - Perform code optimizations inside chunks
  - Since chunk may repeatedly fail, also prepare "Safe Version" code

- Two key ideas:
  1. Form chunks by including code that can be optimized [MICRO-09]
     - E.g.: Multiple low contention critical sections
  2. Cut chunk at points with likely inter-thread communication [This work]

Example of Compiler Optimization: *Barnes*

Lock elision

```
while(flag){
  ....
  if(...){
    Lock(&CellLock→CL[mynode])
      ....
    Unlock(&CellLock→CL[mynode])
  }
  if(...){
    Lock(&CellLock→CL[mynode])
      ....
    Unlock(&CellLock→CL[mynode])
  }

  if(...){
    mynode = ...
  }
}
```

```
while(flag){
  ....
  if(...){
    while(&CellLock→CL[mynode]){}
      ....
  }
  if(...){
    while(&CellLock→CL[mynode])
      ....
  }

  if(...){
    mynode =
  }
}
```

Loop-invariant code motion of lock address generation (part)

Example of Compiler Optimization: *Barnes*

```
Reg=Address_lock_array
while(flag){
  ....
  if(...){
    while(Reg[mynode]){}
      ....
  }
  if(...){
    while(Reg[mynode])
      ....
  }

  if(...){
    mynode = ...
  }
}
```

# Instructions	UnOpt.	Opt.
Dynamic Total	9,306	7,776

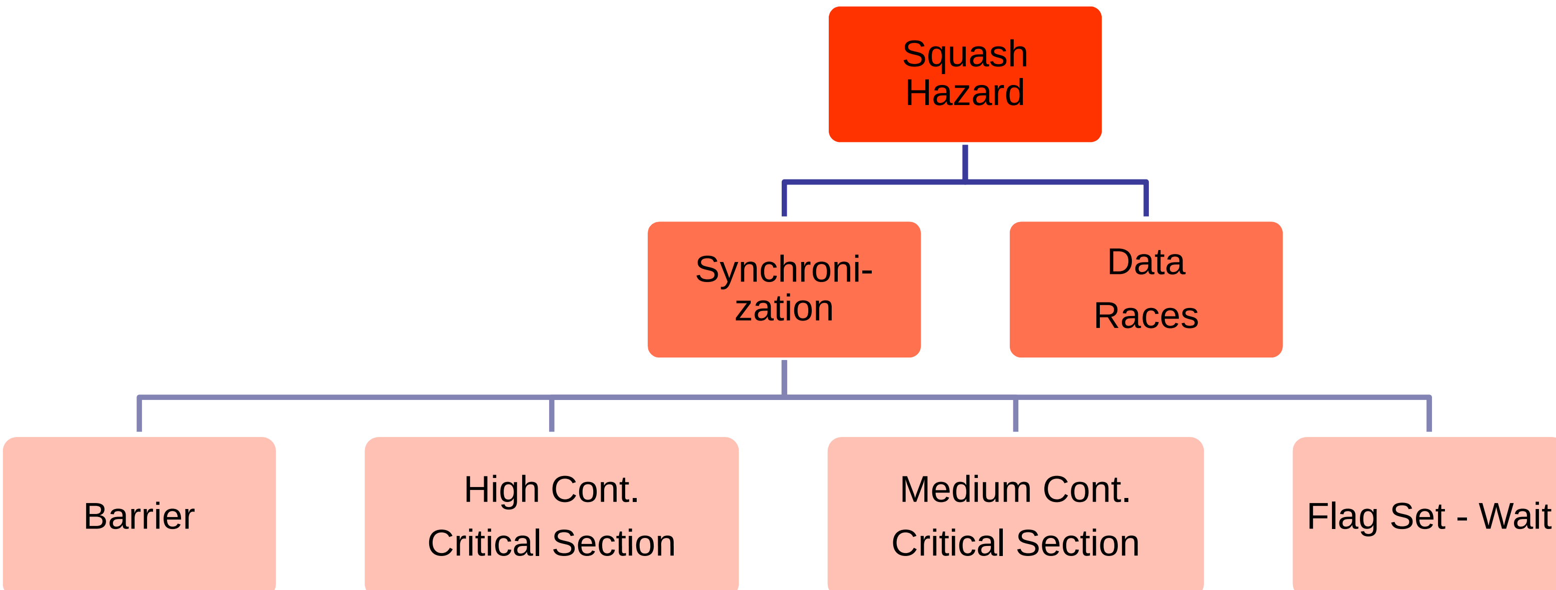
Large chunks are beneficial

How to Cut Chunks to Minimize Squashing?

- Find and characterize Squash Hazards:
  - Operations that frequently cause squashes
  - Intuitively: first communication in a code region with multiple communic
  - Typically: synchronizations and data races; not shared data accesses
  - Use dynamic compilation or a profiling pass

- Transform code with tailored squash-removing algorithms
  - Goal: prevent concurrently-executing chunks from communicating
  - Typically: cutting the chunk before the hazard

Squash Hazards



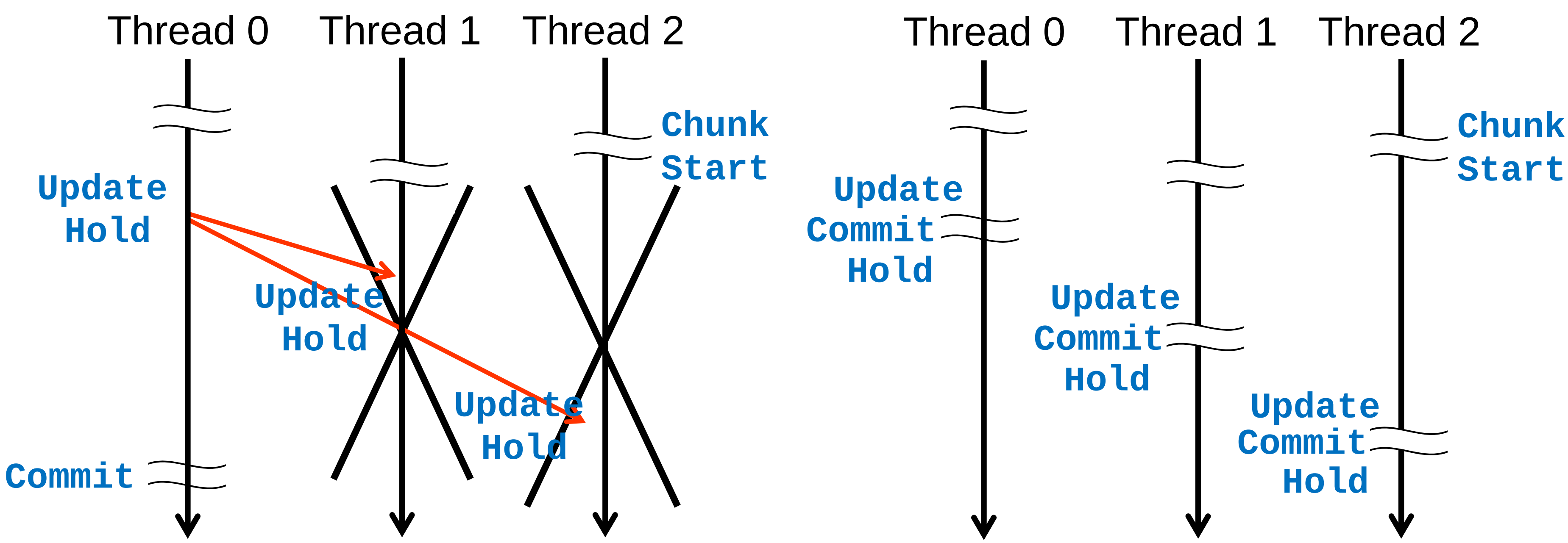
Barriers

```
         ┌ Acquire(lock)
         │ count++
         │ if(count ≥ numProc)
Update  ─┤     count ← 0
         │     flag ← true
         │ end if
         └ Release(lock)
         ┌ while(!flag) do
Hold    ─┤     wait
         └ End while
```

- Barrier leads to large number of squashes
  - Conflicts on **lock**, **count**, and **flag** variable
  - Work before barrier gets lost

Load-Imbalanced Barriers

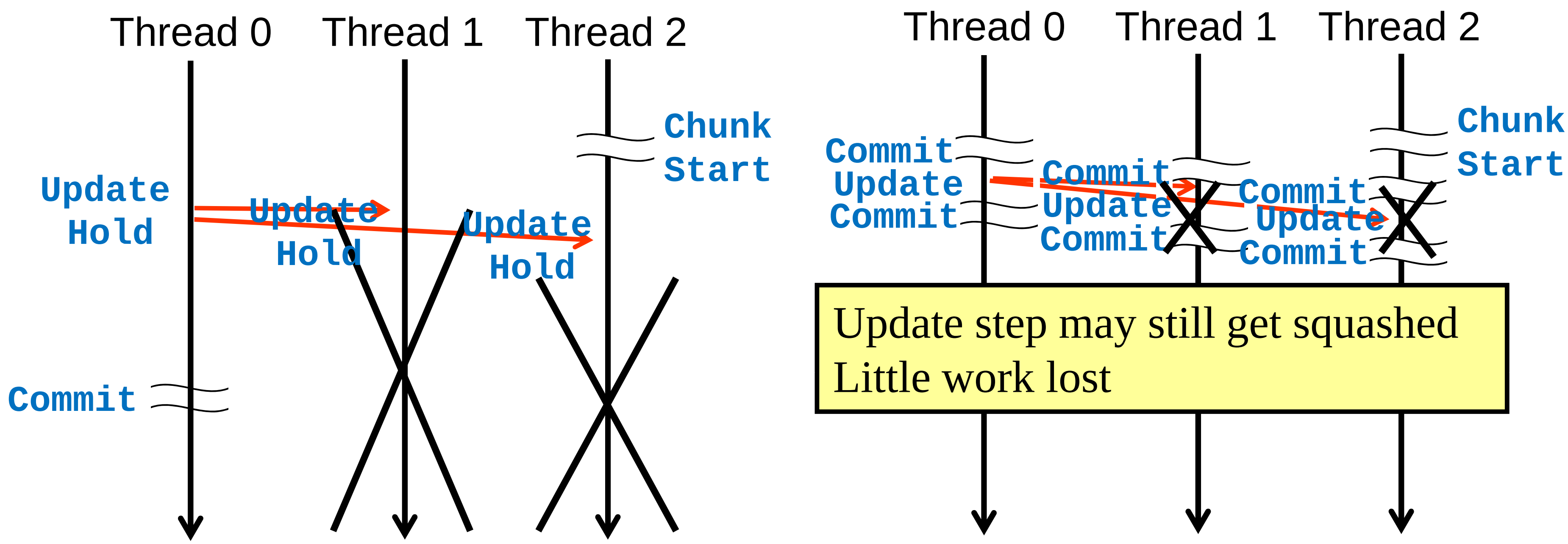
Threads reach barrier at different times



- Solution: Insert a commit after the Update
  - Update immediately visible. Reduces time thread is vulnerable

Load-Balanced Barriers

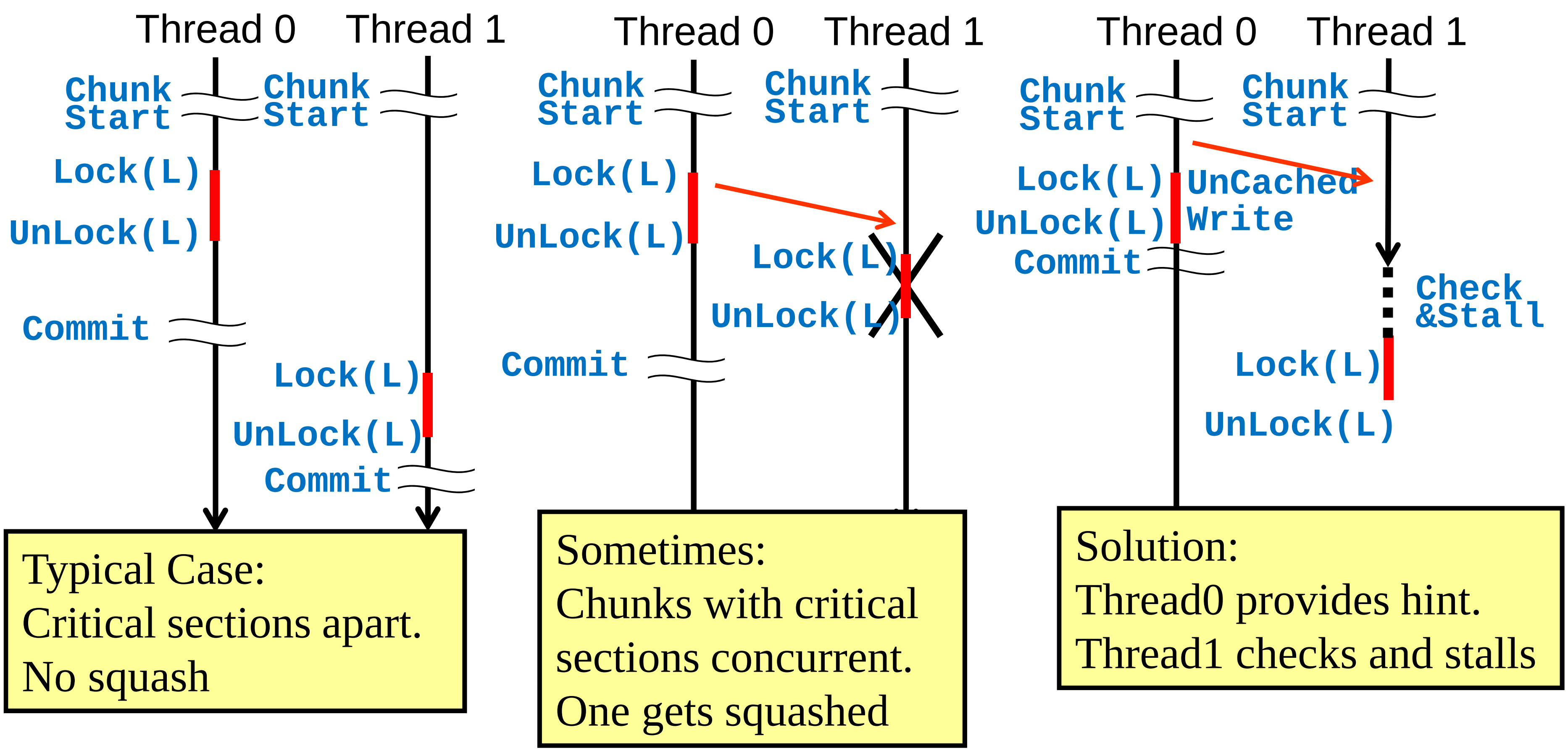
Threads reach barrier almost at same time



- Solution: Insert a commit before and after the Update
  - Saves work before barrier and makes the update visible

Check&Stall Algorithm

Two critical sections that repeatedly communicate (e.g. producer-consumer)



Hidden Inside Synchronization Macros

```
BARRIER:
  Commit
  Update
  Commit
  Hold
```
(a) Balanced Barrier

```
BARRIER:
  Update
  Commit
  Hold
```
(b) Imbalanced Barrier

```
ACQUIRE:
  Commit
  While(lock==taken) {}

RELEASE:
  Commit
```
(c) Lock Elision

```
ACQUIRE:
  Commit
  if(Squashed)
    Stall(Duration)
  Acquire

RELEASE:
  Release
  Commit
```
(d): High Cont. CS

```
ACQUIRE:
  if(Commitflag)
    Commit
  Acquire

RELEASE:
  Release
  if(Commitflag)
    Commit
```
(e) Call−Path Commit

```
ACQUIRE:
  Check&Stall(addr, T1−T0)
  Acquire

RELEASE:
  Release
```
(f) Check&Stall [i]

```
ACQUIRE:
  UncachedWrite(addr,time)
  Acquire

RELEASE:
  Release
  Commit
  UncachedWrite(addr,clear)
```
(g) Check&Stall [j]

```
ACQUIRE:
  Acquire
RELEASE:
  Release

After the Release, periodically
Check&Commit(addr)
```
(h) Check&Commit [i]

```
Earlier: UncachedWrite(addr,set)
ACQUIRE:
  Commit
  Acquire

RELEASE:
  Release
  Commit
  UncachedWrite(addr,clear)
```
(i) Check&Commit [j]

```
SET:
  Set(flag)
  Commit
```
(j) Set

```
WAIT:
  if( !flag ) {
    Commit
    while ( !flag ) {}
  }
```
(k) Wait with No Work

```
WAIT:
  if( !flag ) {
    Commit
    while ( !flag ) {
      Work
      Commit
    }
  }
```
(l) Wait with Large Work

```
WAIT:
  if( !flag ) {
    Commit
    iter = 0
    while ( !flag) {
      iter++
      Work
      if( iter MOD Num == 0 ) {
        Commit
      }
    }
  }
```
(m) Wait with Small Work

- No need to change the source code

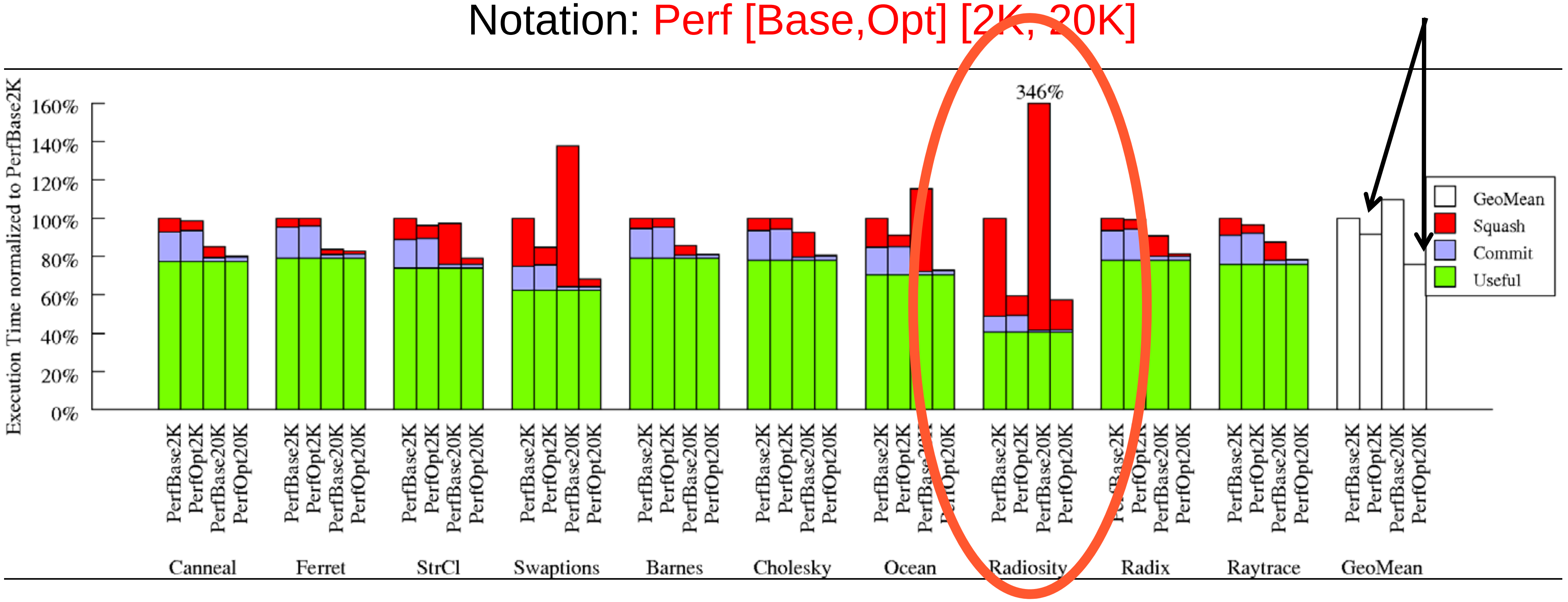
Evaluation

- FlexBulk is a profiling pass
  - Annotates the code to count squashes and commits
  - Characterizes squash hazards and uses best algorithms
- Evaluation based on Pin and SESC simulator
- Apps from PARSEC and SPLASH2; use a training and a deployment input
- Model multicore with 16 cores
  - Blocked execution with lazy commit like Bulk Multicore [CommACM09]
  - Target chunk sizes of 20K instructions → Obtain average of 17K
  - Compare to baseline of 2K chunks
  - Per-core L2: 256KB, 8-way, 32B lines + 64-entry victim cache

Execution Time in Perfect Environment

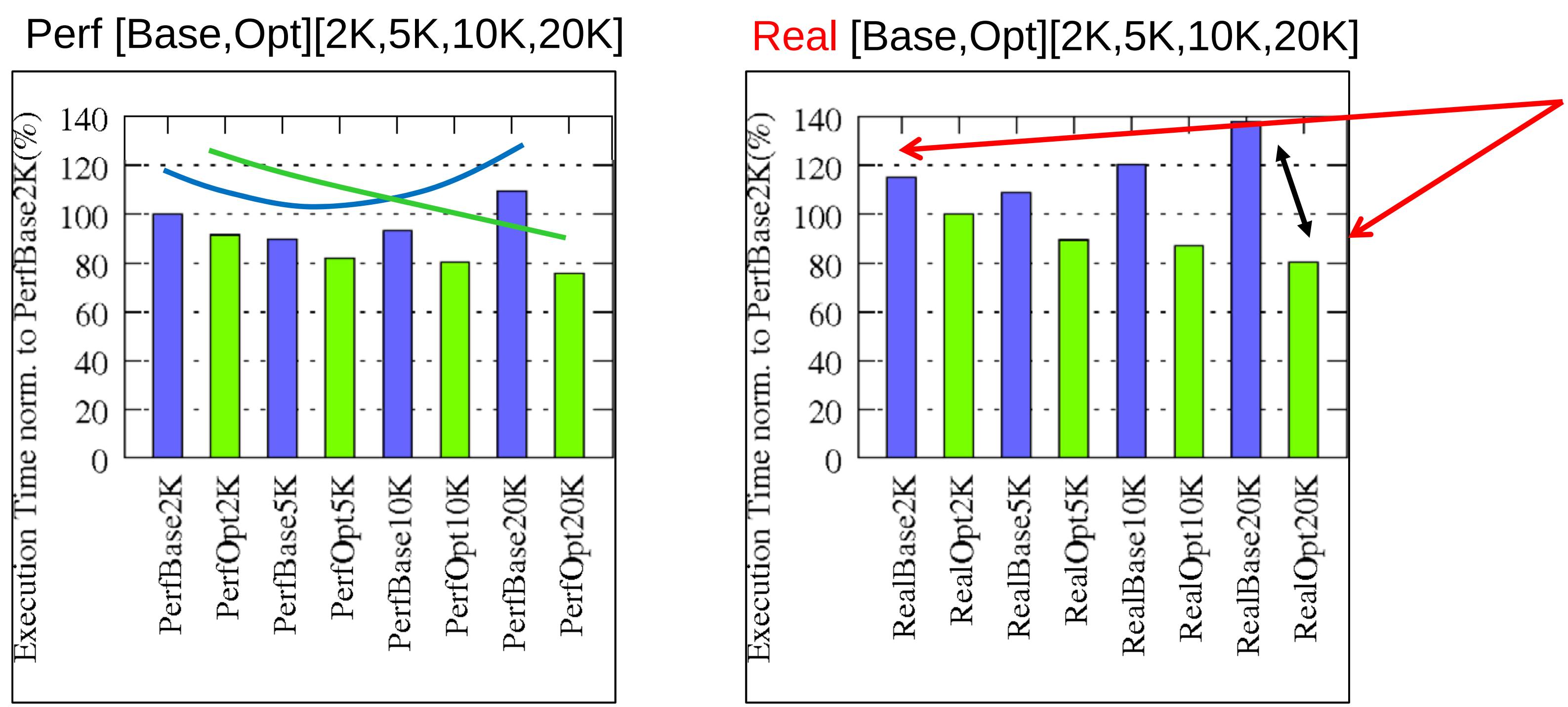
No squashes due to false sharing, false positives in signature or cache overflow

Notation: Perf [Base,Opt] [2K, 20K]



- FlexBulk eliminates most of the squash time
  - For 20K target chunks: Squash 25% → 4%
- PerfOpt20K is 17% faster than PerfOpt2K: low commit + low squash

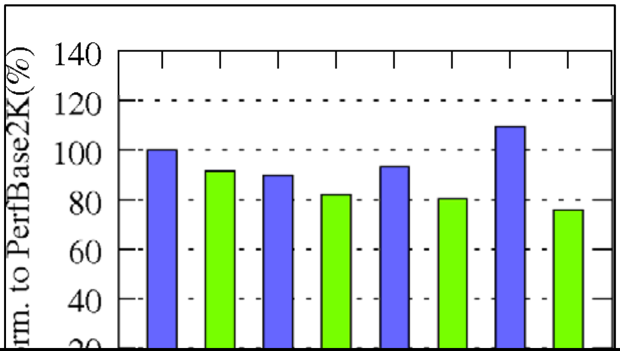
Execution Time for Various Target Chunk Sizes



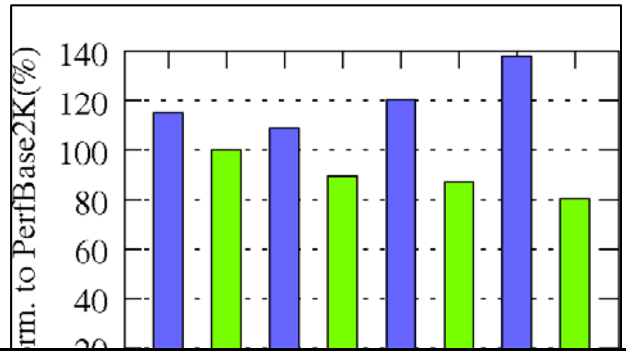
- FlexBulk moves the optimal chunk size to larger sizes: Opt: 20K is best
- Real environment: FlexBulk optimization even more important
- Speedup RealOpt20K over RealBase2K: 1.43x

Execution Time for Various Target Chunk Sizes

Perf [Base,Opt][2K,5K,10K,20K]



Real [Base,Opt][2K,5K,10K,20K]



We have prepared big chunks for the compiler to take advantage of !

- Fle [illegible] best
- Real environment: FlexBulk optimization even more important
- Speedup RealOpt20K over RealBase2K: 1.43x

Conclusions

- FlexBulk: Intelligently form large chunks while minimizing squashes
  - A (mostly) software framework that profiles & transforms the code
  - Characterized Squash Hazards
  - Proposed squash-removing algorithms tailored to them

- Results for 16-processor runs
  - Eliminate 90% of the squash time for 17K-instruction chunks
  - Avg speedup of 1.43x compared to unoptimized 2K-instruction chunks

- Next step: Apply novel compiler optimizations inside these large chunks
  - Pointer-rich codes that cannot be analyzed

FlexBulk: Intelligently Forming Atomic Blocks in Blocked-Execution Multiprocessors to Minimize Squashes

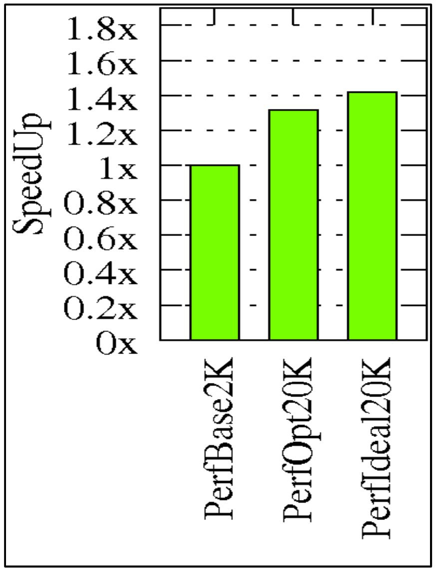
Rishi Agarwal and Josep Torrellas

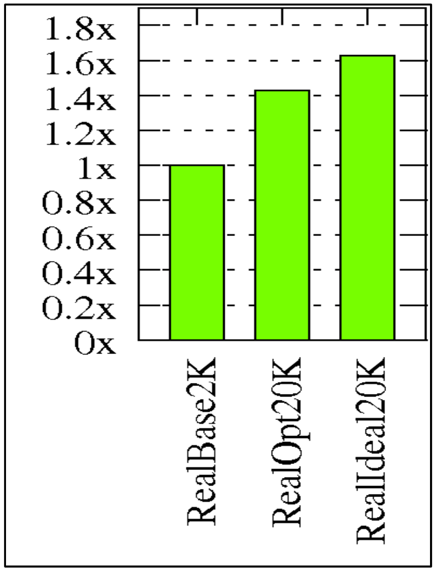
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

Software Interface

- **`Commit`**
  - Processor finishes and commits the current chunk
  - Triggers a register checkpoint and starts a new chunk
- **`Stall(Duration)`**
  - Processor stalls for a number of cycles
- **`Check&Commit(Condition)`**
  - Processor checks Condition and, if true, commits current chunk
  - Non-atomic combination of multiple instructions
- **`Check&Stall(Condition,Duration)`**
  - Processor checks Condition and, if true, stalls for number of cycles
  - Non-atomic as well

Results: Speedup for Perfect, False Sharing and False Positives





- FlexBulk applied to 20K target chunks attains an average application speedup of 1.32x, and 1.43x for the Perf and Real environments
- If we could magically eliminate all of the squash and commit overhead, we would attain only modestly higher speedups, namely, 1.42x, and 1.62x, respectively.
  - Consequently, our optimizations represent a good design point.