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).

```plaintext
P1  P2

st A
ld C
st C
st D

[Bulk: CommACM 09]

[ ] [ ]

st A
st C
ld D
st X
```

- 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

R. Agarwal and J. Torrellas
FlexBulk: Forming Atomic Blocks
Compiler-Driven Chunks: Chunk Size Trade-offs

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

![Graph showing execution overhead vs chunk size with three lines: Squash+Commit, Squash, Commit. Our target is close to the Squash+Commit line near 20 K chunk size.]
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*

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Total</td>
<td>9,306</td>
<td>7,776</td>
</tr>
</tbody>
</table>

Large chunks are beneficial

R. Agarwal and J. Torrellas
FlexBulk: Forming Atomic Blocks
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

Squash Hazard

- Synchronization
  - Barrier
  - High Cont. Critical Section
- Data Races
  - Medium Cont. Critical Section
  - Flag Set - Wait

R. Agarwal and J. Torrellas
FlexBulk: Forming Atomic Blocks
Barriers

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

```python
Acquire(lock)
count++
if(count ≥ numProc)
count ← 0
flag ← true
end if
Release(lock)
while(!flag) do
  wait
End while
```
Load-Imbalanced Barriers

Threads reach barrier at different times

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

R. Agarwal and J. Torrellas
FlexBulk: Forming Atomic Blocks
Load-Balanced Barriers

Threads reach barrier almost at the same time

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

R. Agarwal and J. Torrellas
FlexBulk: Forming Atomic Blocks
Check&Stall Algorithm

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

Typical Case: Critical sections apart. No squash

Sometimes: Chunks with critical sections concurrent. One gets squashed

Solution: Thread0 provides hint. Thread1 checks and stalls

R. Agarwal and J.Torrellas
FlexBulk: Forming Atomic Blocks
Hidden Inside Synchronization Macros

<table>
<thead>
<tr>
<th>BARRIER:</th>
<th>ACQUIRE:</th>
<th>RELEASE:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commit</td>
<td>Commit</td>
<td>Release</td>
</tr>
<tr>
<td>Update</td>
<td>While(lock==taken) {}</td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td>if(Squashed)</td>
<td>if(Commitflag)</td>
</tr>
<tr>
<td>Hold</td>
<td>Stall(Duration)</td>
<td>Commit Acquire</td>
</tr>
<tr>
<td>Hold</td>
<td></td>
<td>Release</td>
</tr>
<tr>
<td>(a) Balanced</td>
<td>(c) Lock Elision</td>
<td>(f) Check &amp; Stall [i]</td>
</tr>
<tr>
<td>Barrier</td>
<td>(d) High Cont. CS</td>
<td></td>
</tr>
<tr>
<td>(b) Imbalanced</td>
<td>(e) Call-Path Commit</td>
<td></td>
</tr>
<tr>
<td>Barrier</td>
<td></td>
<td></td>
</tr>
<tr>
<td>After the Release, periodically Check &amp; Commit(addr)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(h) Check &amp; Commit [i]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACQUIRE:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acquire</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RELEASE:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Release</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(g) Check &amp; Stall [j]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- No need to change the source code

R. Agarwal and J. Torrellas
FlexBulk: Forming Atomic Blocks
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

R. Agarwal and J. Torrellas
FlexBulk: Forming Atomic Blocks
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

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

- 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

R. Agarwal and J. Torrellas
FlexBulk: Forming Atomic Blocks
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.