Speculative Multiprocessing

Alex Nizhner, Yang Gu

The Challenge

- Efficiently parallelizing arbitrary C programs
  - Complicated control flow
  - Complicated memory access patterns
- The burden is typically on the programmer
  - Identify independent regions (threads) that can be safely executed in parallel
  - Explicit synchronization
- Compilers can’t identify dependencies between loads and stores

The Idea

- “Oblivious” Parallelization
  - Divide the program up arbitrarily
  - Execute sub-regions in parallel
  - Speculate on the contents of memory

Papers

- J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai and Todd C. Mowry. *A Scalable Approach to Thread-Level Speculation*, CMU
“Oblivious” Parallelization

```c
for (i=0; i < n ; i ++)
{
    ...
    ...
    ...
    Body(i)
}
```

What makes this difficult?

- RAW Hazards!
- “earlier” regions write memory read by “later” regions
- Speculation fails

RAW Hazards

```c
for(i=0; i < n ; i++)
{
    Body(i)
}
```

Doing it Right

- Detect the violations
- Deal with the violations
- Ensure memory consistency
Detecting Violations

Finishing Touches
- Dealing with violations
  - Speculation must fail
  - Threads “squashed” and restarted
- Memory consistency
  - Speculative threads can’t write to main memory
  - Must buffer outstanding writes
  - Either completes successfully or gets squashed
  - Eventually becomes non-speculative
    - Wait if necessary
    - Epilogue()
    - Safe to commit

Approach 1: Stanford
- Single-Chip multiprocessor
- Detection
  - Write-through coherence
  - All writes globally visible (“Write bus”)
  - Invalidation or violation detection on each write
- Consistency
  - Write buffers
  - Records speculatively-modified cache lines
  - Drained to L2 cache when safe
  - Drains a byte at a time, when L2 cache is free

Approach 2: CMU
- Works on most MP architectures
  - Interconnection network
  - Main memory
- Invalidation-based coherence
  - Detection
    - Main memory controller
    - Invalidation messages sent to CPU’s with local copies
  - Consistency
    - Ownership Required Buffer (ORB)
    - Drained as soon as speculation ends
Basic Example of Violation Detection

Processor 1
Epoch 5

Processor 2
Epoch 6

L1 Cache

Speculatively
Modified?

Speculatively
Loaded?

Violated

Violation ? = False

Violation ? = True

Time

X = 1 → 2

T

T

SL SM

T

T

SL SM

2003-10-13

When Speculation Succeeds

Logical order

Epoch 0

Speculative execution

Pending

commit

Epoch 1

Speculative execution

Pending

commit

Epoch 2

Speculative execution

Commit

Logical order

Waiting for the homefree token

Speculative execution

Speculative execution

Speculative execution

If in SpeS & SM, issue an upgrade-request to acquire exclusive ownership

Once it is owner exclusively, the line may transition to the dirty state

The Speedup: conclusion

- Offers absolute speedups
  - ranging from 8% to 46% on a 4 processor single-chip multiprocessor
  - up to 75% on multi-chip architectures
- Reasonable speedups for up to 4 processors
- Diminishing returns
- Visible degradation under further parallelization!

To Speculate Or Not to Speculate?
## Pros and Cons: increased parallelism

- **Pros**
  - Obvious, when it works

- **Cons**
  - Increased likelihood of hazards
  - Decreased locality!
    - Data distributed
    - Frequent invalidations

## Increased Memory Traffic

- **Why?**
  - Software overhead
    - Thread/epoch creation
    - Epilogue()
  - Superfluous accesses
    - Failed threads
  - Loop Speculation
    - Can’t put shared variables in registers

## Tradeoffs: Thread Size

- **Long:**
  - Likelihood of hazards increases
  - More writes to buffer
    - Potential for costly overflows

- **Short:**
  - Overhead becomes prohibitive

## To Speculate Or Not to Speculate?

- **Violation counters**
  - Avoid speculating regions with many dependencies

- **Epoch timers**
  - Filter out epochs that are too short or too long
Questions?

😊

Thank you!