Speculative Multi-threading

Brad Schumitsch
Amin Firoozshahian

EE392C, Lecture 4
Spring 2003
Outline

- What is speculation?
- What are the requirements?
- First paper:
  - Taxonomy
  - Different schemes, advantages, disadvantages
- Second paper:
  - Can it be done purely by software?
- Discussion
What Is Speculation?

Problem:
- Traditional auto-parallelization is limited
  - Many applications are hard to parallelize or not parallelizable

Solution:
- "ASSUME" that the code is parallel and run it
- "DETECT" when anything goes wrong and roll back
What Is Speculation? (Continued)

Sequential code

Parallellized code

1

2

3

It’s an easy way to parallalize code!
Requirements

Sequential Code

Section i
read X
write X

Section i+1
read X
read X
write X
read X

Speculatively Parallelized Code

Section i
read X
write X

Section i+1
read X
read X
write X
read X

VIOLATION!

FORWARDING

Ability to detect violations
Ability to forward data (optimization)
Requirements (Continued)

- Ability to discard unsafe state
- Commit changes in order
Taxonomy Paper

- Introduces a taxonomy for different speculative schemes.
  - Categorization of memory state buffering
  - How fast to commit changes

- Attempts to quantify how effective these different schemes are.
Architectural Main Memory

- Main memory contains committed states
- Speculative states kept in caches or buffers
- Relatively fast squashes, slow commits
Eager vs. Lazy AMM

- In eager, main memory is updated immediately after thread commits.

- Lazy AMM allows immediate passing of head token.
Future Main Memory

Main memory contains the most recent version of each variable.

- Need complicated mechanism (such as logs) to enable rollbacks
- Commits are faster, squashes slower
Single vs. Multiple Tasks

- **Single Task**
  - Need to stall after task finished until become head
  - Relatively simple hardware

- **Multiple Tasks**
  - Keeping processors as busy as possible
  - Load balancing is less important
  - Complicated hardware

- **Multiple Tasks, single version**
  - Have to stall when accessing same variable
Comparison Results

- Lazy commits outperform eager commits
- Lazy vs. FMM, depends on application
- Multiple Versions outperform single versions of a task
- Differences in CMP becomes small
Software Paper

Relying on compiler to distinguish:
- “Private” references
- “Loop-carried” references
- “Ambiguous” references

Rely on RAW’s low latency network
- Replace memory access instructions by communication instructions
Software Paper (Continued)

“Memory nodes” check for violations
Uses logs for rolling back the safe state

WEAKNESS:

- No analytical performance numbers
- Not enough application studies

Question: Is this approach useful if we don’t have a low latency network on chip?
Discussion

Hardware/software trade off: where shall we draw the line?
- How much hardware do we need?

How compiler can help?
- Can it detect possible threads?
- Can it minimize “ambiguous” memory references?
Discussion (Continued)

Which parts of the code are good for speculative parallelization?
- Loops?
- Procedure calls?
- Conditional statements?

How scalable the approach might be?
- How performance / complexity might scale with number of threads?
What shall be granularity of speculative state?
- Cache line
- Byte

Which type of applications are good candidates for speculative parallelization?
Any support for speculation at the programming level?

Speculative contexts?

Is it a good idea if we have a region of memory as “Transactional Memory”? 