An Optimistic Perspective on Speculative Multithreading

Chris Pickett, Richard Halpert
Haiying Xu, Clark Verbrugge
School of Computer Science, McGill University
{cpicke,rhalpe,hxu31,clump}@sable.mcgill.ca

Allan Kielstra
IBM Toronto Lab
kielstra@ca.ibm.com

October 18th, 2006

Challenges for Parallel Computing Workshop, CASCON 2006
1 Introduction

2 General Optimisations

3 Speculative Locking and Transactions

4 Purity Analysis

5 Conclusions
Speculative Multithreading

(a) INVOKE<X>  <X>RETURN

(b) INVOKE<X>  fork  join  T2

Legend:
- white: pre-invoke instructions
- gray: method body
- black: post-invoke instructions
"Speedup"

The image shows a bar chart with the y-axis labeled "relative speedup factor" and the x-axis labeled "slowdown" with values 3x, 5x, 4x, 5x, 2x, 10x, 11x, and 4x. The chart compares various factors affecting speedup:

- Orange bars: no method entry or exit
- Purple bars: no dependence buffering
- Green bars: no return value prediction
- Blue bars: full runtime TLS support
- Yellow bars: no object allocation

Each category is represented by bars for different slowdown values, showing the relative speedup factor for each condition.
Return value prediction (RVP) is important.

Unfortunately, overheads are high.

Possible solutions:
  - Update predictors lazily
  - Disable hybrid sub-predictors dynamically

Both approaches compromise accuracy.
Currently, threads are only created at non-speculative callsites.

Indeed, helper threads are most often idle.

Solution: populate queue by allowing nested speculation.

Problems:
- Memory management
- Aborting trees of children
- Profiling and statistics
Currently, we fork a child thread at every callsite. Clearly this is inefficient!

Several hardware proposals select fork points dynamically. Fork heuristic inputs:
- Dependence buffer size
- Child call stack size
- Successful commit history
- Historical child and parent thread lengths
  - Measured in instructions, bytecodes, or cycles.

Sadly, reduction in overhead makes experimentation difficult.
Outline

1. Introduction
2. General Optimisations
3. Speculative Locking and Transactions
4. Purity Analysis
5. Conclusions
Results show synchronization and memory constraints are important.

Speculative locking is fairly straightforward:
- Allow speculative threads to acquire locks.
- Allow non-speculative threads to become speculative.
- Replay lock operations at commit time.

Previous work has not shown great benefits.
Transactional programming is a hot field right now.

atomic \{ ... \} replaces synchronized (o) \{ ... \}

- Deadlock goes away!

Pessimistic transactions $\equiv$ one global lock

Optimistic transactions: speculatively enter atomic blocks

- Essentially pessimistic transactions + speculative locking
Transactional Lock Allocation

- Take regular, correctly synchronized (race-free) program.
- Convert all lock operations to use a single global lock.
- Convert all notify()’s to notifyAll()’s.
  - Voilà, pessimistic transactions.
- Transactional lock allocation: use pointer analysis to split global lock into multiple externally transactional locks.
- Experiments:
  - Original program, (+/−) speculative locking
  - One global lock, (+/−) speculative locking
  - Transactional lock allocation, (+/−) speculative locking
One of our return value predictors is memoization-based.

Why not use it for non-speculative Java memoization?

Need some definition of purity:
  
  What qualifies a method as being safe to skip?
Strong purity: method does not access heap, does not call native methods, does not throw an exception.

We can weaken this in small steps:
- Allow for object allocation, provided object does not escape.
- Allow for heap r/w to locally allocated objects
- Allow for exceptions, provided they are caught.

Now we can weaken this interprocedurally:
- Allow for escaping objects and exceptions, provided at some outer method they do not escape.
- This outer method is pure and memoizable.
Observation: pure methods do not conflict with speculation.
  - Weaken purity definition again to allow heap reads.
  - Now a pure callsite becomes a good fork point.

Problem:
  - May be redundant information with respect to fork heuristics.
  - Nevertheless, infrastructure is valuable for profiling.
How can we detect purity?

Static analysis: complicated, expensive, conservative, and generally well-studied.

Dynamic analysis: once pure for a given set of parameters, a method is always pure.
  - Instrumentation is probably too expensive, might need a sampling-based approach.
Conclusions

1 Introduction

2 General Optimisations

3 Speculative Locking and Transactions

4 Purity Analysis

5 Conclusions
Speculative multithreading is alluring but expensive!

There exist multiple new research opportunities:
- Fork heuristics, lazier value prediction, nested speculation.
- Speculative locking, transactional lock allocation.
- Dynamic purity analysis.