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

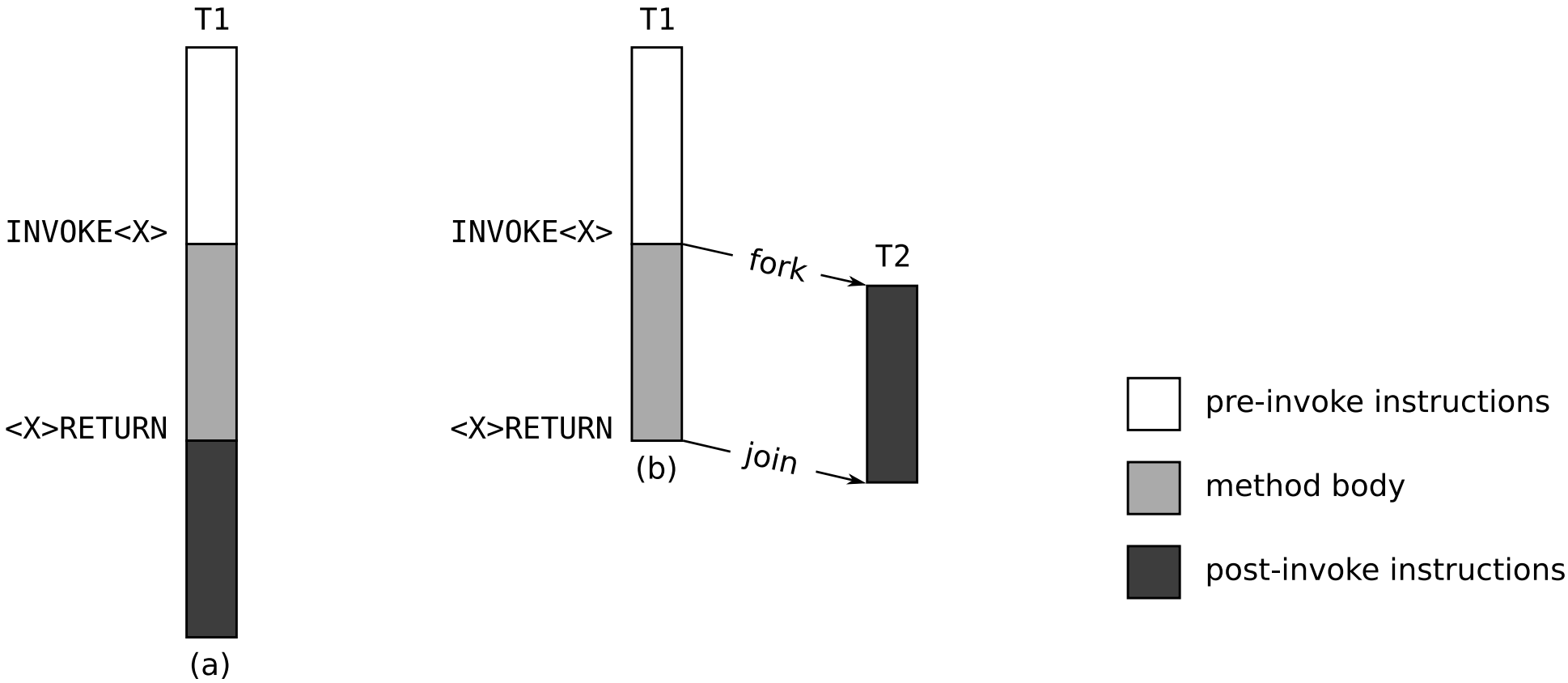
Allan Kielstra
IBM Toronto Lab
kielstra@ca.ibm.com

October 18th, 2006

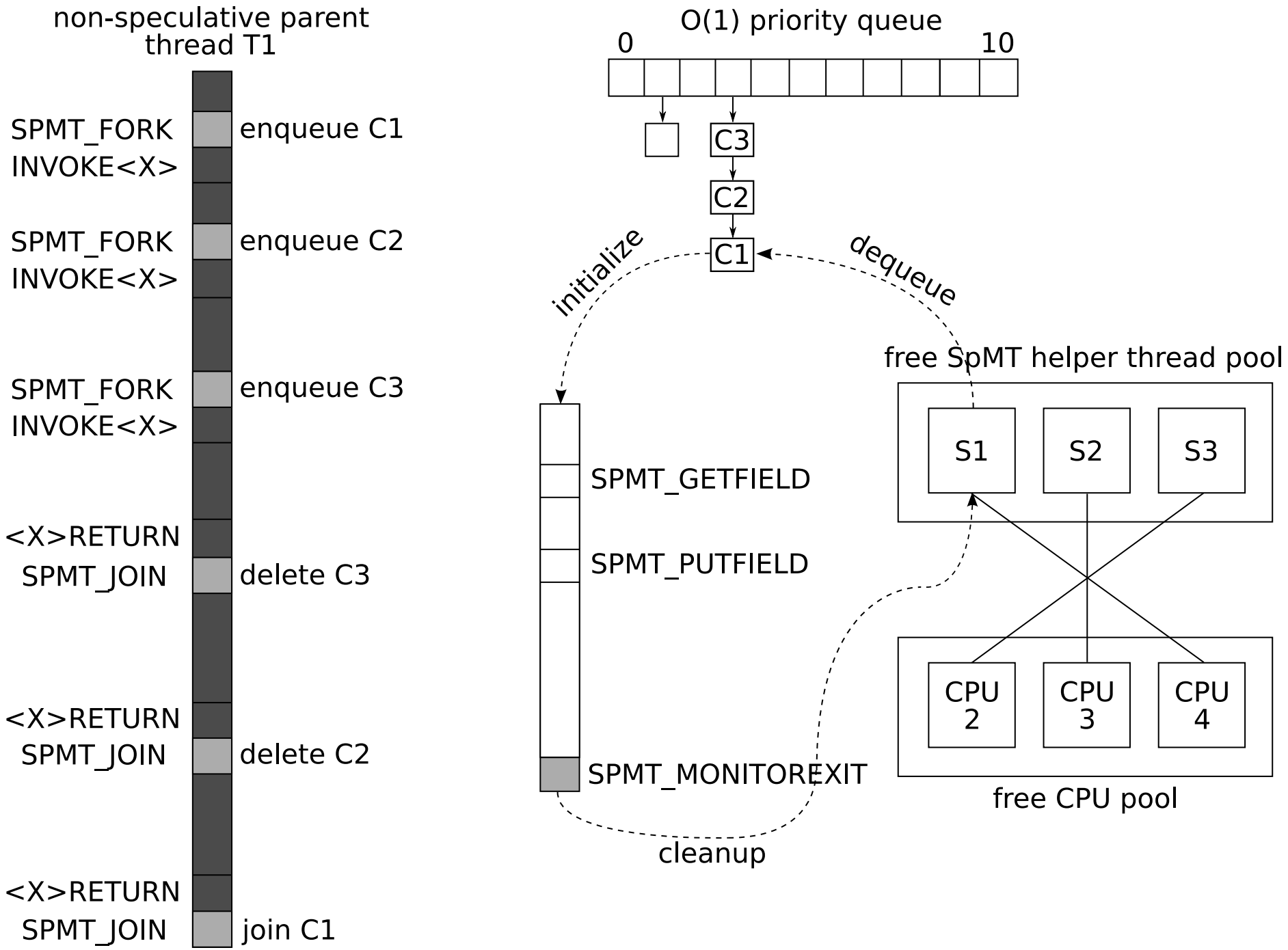
Challenges for Parallel Computing Workshop, CASCON 2006

Outline

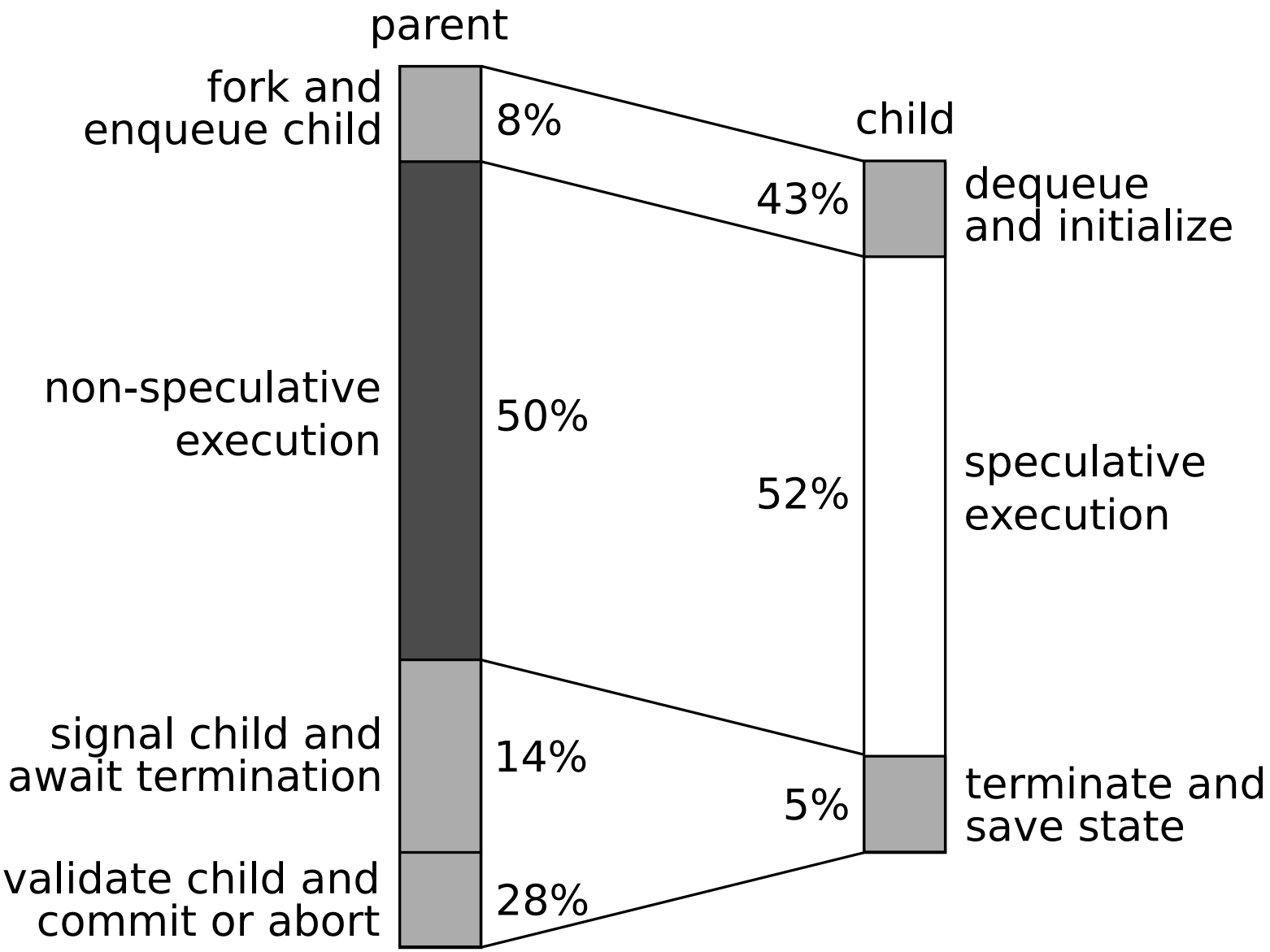
Speculative Multithreading



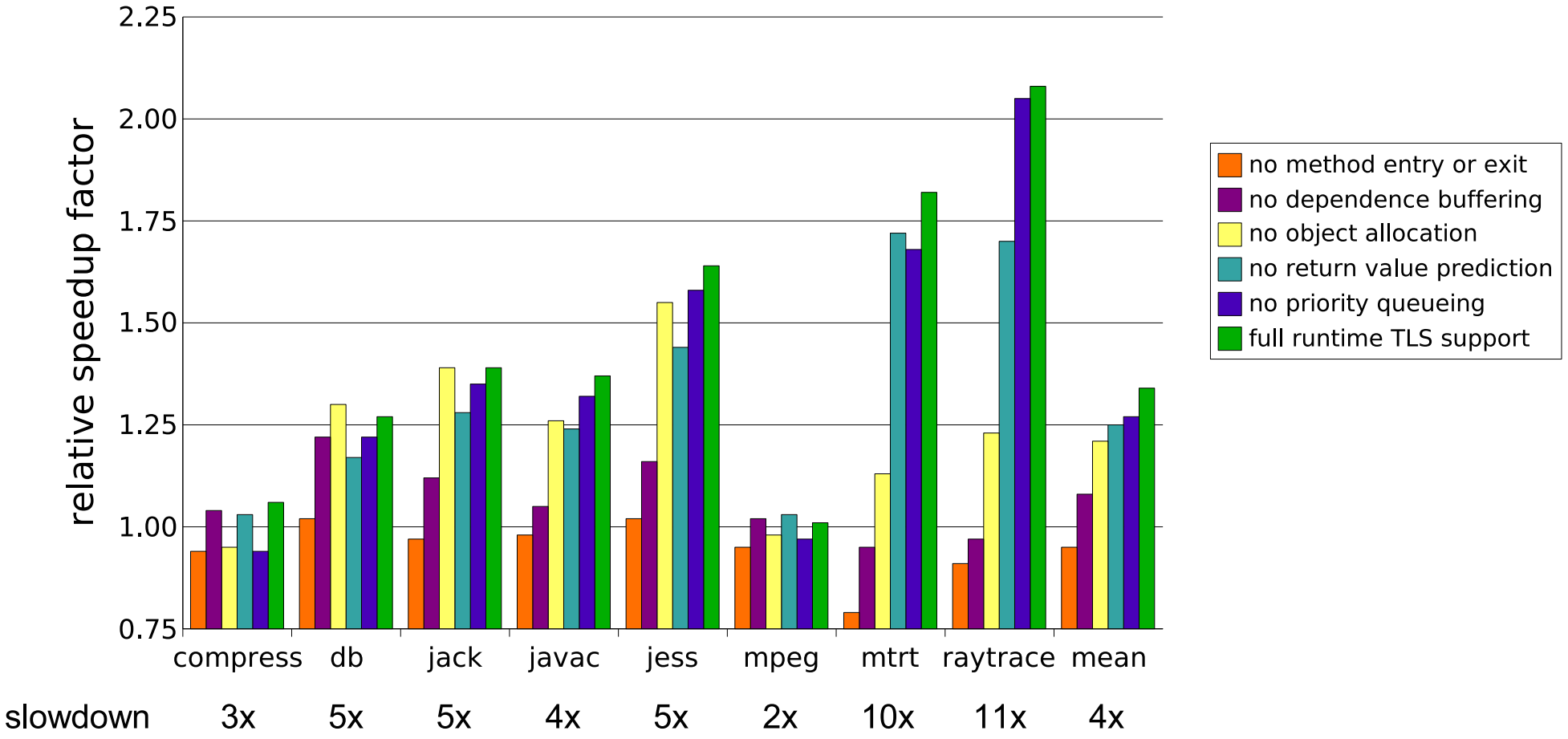
SableSpMT Implementation



Overhead


parent
fork and
enqueue child
8%
child
43%
dequeue
and initialize
non-speculative
execution
50%
52%
speculative
execution
signal child and
await termination
14%
5%
terminate and
save state
validate child and
commit or abort
28%

“Speedup”


relative speedup factor
2.25
2.00
1.75
1.50
1.25
1.00
0.75
compress
db
jack
javac
jess
mpeg
mtrt
raytrace
mean
slowdown
3x
5x
5x
4x
5x
2x
10x
11x
4x
no method entry or exit
no dependence buffering
no object allocation
no return value prediction
no priority queueing
full runtime TLS support

Outline

Return Value Prediction

- Return value prediction (RVP) is important.
- Unfortunately, overheads are high.
- Possible solutions:
  - Update predictors lazily
  - Disable hybrid sub-predictors dynamically
- Both approaches compromise accuracy.

Nested Speculation

- 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

Fork Heuristics

- 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

Speculative Locking

- 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.

Transactions

- 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

Outline

Non-speculative Memoization

- 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?

Purity Gradations

- 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.

Speculation and Purity

- 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.

Dynamic Purity Analysis

- 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

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.