Dynamic Performance Tuning for Speculative Threads

Yangchun Luo, Venkatesan Packirisamy, Nikhil Mungre†, Ankit Tarkas†, Wei-Chung Hsu, and Antonia Zhai

Dept. of Computer Science and Engineering
†Dept. of Electronic Computer Engineering
University of Minnesota – Twin Cities
Motivation

Multicore processors brought forth large potentials of computing power

Exploiting these potentials demands thread-level parallelism
Exploiting Thread-Level Parallelism

Sequential

Traditional Parallelization

Thread-Level Speculation (TLS)

Potentially more parallelism with speculation

But Unpredictable
Addressing Unpredictability

State-of-the-art approach:
- Profiling-directed optimization
- Compiler analysis

A typical compilation framework:

- Profiling run
- Feedback
- Profiling info
- Source code
  - Region selection
  - Parallel code generation
  - Speculative code optimization
  - Machine code
Problem #1

Hard to model TLS overhead statically, even with profiling
Profile-based decision may not be appropriate for actual input
Problem #3

Behavior of the same code changes over time
Problem #4

**Extra misses**

**Sequential**

- Load 0x22: miss
- Load 0x22: hit

**Parallel**

- Load 0x22: miss
- Load 0x22: miss

**Prefetching**

- Load *p* = 0x88

**Pollution**

- Load *p* = 0x80
- Load *p' = 0x08

How important are cache impacts?
Impact on Cache Performance

A major loop from benchmark art (scanner.c:594)

Cache impact is difficult to model statically

Cache: p == p'
Prefetched (+)

load *p=0x88

load *p'=0x88

2.5X

60%
Our proposal

Proposed Performance Tuning Framework

Compiler Analysis
- Source code
- Region selection
- Parallel code generation
- Speculative code optimization
- Machine code

Runtime Analysis
- Performance Evaluation
- Runtime Information
- Dynamic Execution
- Performance Profile Table
- Tuning Policy

Adaptations for inputs and phase changes

Accurate Timing & Cache Impacts

Profiling info

Dynamic Execution

Compiler Annotation
Parallelization Decisions at Runtime

Thread spawning

End of a parallel execution

Performance Profile Table

parallelize

spawn

serialize

spawn

spawn

spawn

Parallel execution

Sequential execution

Parallel Execution

Evaluate performance

New execution model facilitates dynamic tuning
Evaluating Performance Impact

Runtime Analysis

Simple metric:
cost of speculative failure

Comprehensive evaluation:
sequential performance prediction
Sequential Performance Prediction

Hardware Counters

iFetch  ExeStall
Busy       Cache

Programmable Cycle Classifier

Information from ROB

tick

Performance Impact

TLS overhead  ExeStall  iFetch  Busy

Speculative Parallel  Predict  Predicted sequential
Cache Behavior: Scenario #1

**TLS**
- **T0**
  - load p (unmodified) → miss

**SEQ**
- **T0**
  - load p → hit
  - load p → miss

**Predicted**
- **T0**
  - load p → hit

**Count** the miss in squashed execution
Cache Behavior: Scenario #2

Not count a miss if:
- tag matched
- invalid status
Tune the performance

Runtime Analysis

Performance Evaluation

Dynamic Execution

Performance Profile Table

Tuning Policy

Exhaustive search

Prioritized search
How to prioritize the search

Search order:
- Inside-Out
- Compiler suggestion
- Outside-In

Performance summary metric:
- Speedup or not?
- Improvement in cycles

Performance Summary metric: Speedup or not? Improve. in cycles

Search order: Suboptimal

Quantitative

Simple

Outside

Compiler suggestion

Quantitative + StaticHint

More Aggressive

Inside-Out

Search order
Evaluation Infrastructure

Benchmarks
- SPEC2000 written in C, -O3 optimization

Underlying architecture
- 4-core, chip-multiprocessor (CMP)
- speculation supported by coherence

Simulator
- Superscalar with detailed memory model
- simulates communication latency
- models bandwidth and contention

➢ Detailed, cycle-accurate simulation
Comparing Tuning Policies

![Graph showing speedup wrt. SEQ for different policies: Simple, Quantitative, and Quantitative+StaticHint. The graph includes benchmarks like ammp, art, bzip2, crafty, earthquake, gcc, gzip, mcf, mesa, parser, perlbench, twolf, vortex, vpr-p, and vpr-r. The chart highlights speedup factors such as 1.17x, 1.23x, and 1.37x. The section 'Parallel Code Overhead' is also noted.]
Profile-Based vs. Dynamic

Dynamic outperformed static by ≈10% (1.37x/1.25x)

Potentially go up to ≈15% (1.46x/1.27x)
Related works: Region Selection

Statically

- Compiler Heuristics
  [Vijaykumar Micro’98]
- Balanced Min-Cut
  [Johnson PLDI’04]
- Simple Profiling Pass
  [Liu PPoPP’06] (POSH)
- Extensive Profiling
  [Wang LCPC’05]
- Profile-Based Search
  [Johnson PPoPP’07]

Dynamically

- Runtime Loop Detection
  [Tubella HPCA’98]
  [Marcuello ICS’98]
- Saving Power from Useless Respawning
  [Renau ICS’05]

Dynamic Performance Tuning
[Luo ISCA’09] (this work)
Conclusions

Deciding how to extract speculative threads at runtime
- **Quantitatively** summarize performance profile
- **Compiler hints** avoid suboptimal decisions

Compiler and hardware work together.
- **37%** speedup over sequential execution
- **≈10%** speedup over profile-based decisions

*Configurable HPMs enable efficient dynamic optimizations*

**Dynamic performance tuning can improve TLS efficiency**