Speculative Multithreading: from Multiscalar to MSSP

Guri Sohi

Computer Sciences Department
University of Wisconsin-Madison
Outline

• A decade of speculative multithreading evolution
• Multiscalar background and rationale
• Multiscalar
• Parallelization and Speculative Parallelization
• Speculative Data-driven Multithreading (DDMT)
• Speculative Slices
• Master-Slave Speculative Parallelization (MSSP)
Hardware Wish List (circa 1993)

- Use of simple, regular hardware structures
- Clock speeds comparable to single-issue processors
- Easy growth path from one generation to next
  - Reuse existing processing cores to extent possible
  - No centralized bottlenecks
- Exploit available parallelism
Software Wish List (circa 1993)

- Write programs in ordinary languages (e.g. C)
- Target uniform hardware-software interface
  - Facilitate software independence and growth path
- Maintain uniform hardware-software interface, i.e., do not tailor for specific architecture
  - Minimal OS impact
  - Facilitate hardware independence and growth path
- Place few demands on software
  - make minimum requirements for guarantees
The Opportunity and Objective (circa 1993)

• Many tens of millions of transistors on a chip vs. few million today

• Can integrate several (tens?) of today's processors, plus supporting hardware, on a chip

Use available resources to minimize program execution time!
A Bird’s Eye View

- Sequence through static program and establish a window of execution
- Establish dependence relationships within window
- Set up parallel execution schedule for operations in window
- Provide resources to implement parallel execution schedule
Multiscalar Paradigm (Franklin, Breach, Vijaykumar)

- Break sequencing process into two steps
  - Sequence through static representation in *task-sized* steps
- Sequence through each task in conventional manner
- Split large instruction window into ordered tasks
- Assign a task to a simple execution engine; exploit ILP by overlapping execution of multiple tasks
- Use separate PCs to sequence through separate tasks
- Maintain the appearance of a single-PC sequencing through the static representation
Multiscalar Big Picture: Basics

- PROGRAM
- PROC UNIT 1
- PROC UNIT 2
- PROC UNIT 3

Predict A
Predict B
Predict C
Multiscalar Big Picture: Hardware
Dependence Prediction (Moshovos)

- circa 1995-96
- To prevent “over speculation” of dependences
- Predict load-store dependence relationships
  - use to control dependence speculation
- Learn about likely violations and synchronize
- Emerged as key technology
  - useful regardless of parallelism exploitation model
Stepping Back

- Multiprocessor microarchitecture
- What should the “work” be for different processing units?
  - multiscalar is speculative form of traditional parallelization
- Relevance of solutions to other parallelism models
Work for Distributed/Multithreaded Processor

- Independent programs
  - increase overall processing throughput
- Independent threads of multithreaded application
  - increase overall throughput
- Related threads
  - e.g., for reliability
- But what about speeding up single program execution?
  - how to “parallelize” or “multithread” single program?
Program Parallelization

• What does it mean to parallelize?
  ◦ how to divide program into multiple portions

• What constrains parallelization?
  ◦ dependences (especially ambiguous)

• How to overcome constraints?
  ◦ use speculation
Program Parallelization -- Theme 1

• Traditional view: control-driven threads
  - divide work into multiple groups of instructions
    - conservative assumptions about dependences constrain parallelization
  - each group is specified using traditional control-driven (von Neumann) semantics

• A newer view: multiscalar
  - use dependence speculation to overcome constraints
Program Parallelization -- Theme II

• Another traditional view: dataflow
  ◦ divide work into (dependent) computations
  ◦ each computation is represented in a data-driven manner

• A newer view: speculative data-driven “threads”
  ◦ use speculation to facilitate thread creation
Motivation for Data-driven Threads

- Program execution: processing of low-latency instructions, with pauses for high-latency events
- Parallelizing low-latency instructions isn’t crucial
- Overlapping high-latency events is what matters!
- “Threads” should initiate high-latency events early
- Need to “sequence” these instructions early
Speculative Data-Driven Multithreading

- Isolate data-driven threads from program
- Execute isolated (data-driven) threads in parallel
- Also called “pre-execution”

Issues
- nature of threads
- how to launch and execute threads
- communicating values between threads and main program
DDMT (Roth)

**unoptimized execution**
- master thread
- fetch
- execute

```plaintext
R1 = R1 + 1
R1 = R1 + 1
R1 = R1 + 1
R2 = ld [R1]
R1 = R1 + 1
R2 = ld [R1]
R1 = R1 + 1
bz R2, 0x2c
```

**pre-execution**
- master thread
- pre-execution thread
- fetch
- execute
- fetch
- execute

```plaintext
fork
R1 = R1 + 1
R1 = R1 + 1
R1 = R1 + 1
R2 = ld [R1]
R1 = R1 + 1
R2 = ld [R1]
R1 = R1 + 1
R1 = R1 + 1
bz R2, 0x2c
R2 = ld [R1]
R1 = R1 + 1
bz R2, 0x2c
```

- load & branch computation instructions
- memory latency problem load
- branch resolution
- pipeline latency
- speedup
- early resolution

**Problem**
- memory latency
- branch resolution
- pipeline latency

**Speedup**
DDMT, cont.

- Identify problem loads & branches
- **Construct Speculative Data-Driven Threads (DDT):**
  - select a fork point with sufficient latency tolerance
  - profile to identify common data-flow predecessors
  - pack these instructions into static DDT
- **Execute DDTs on idle SMT thread contexts**
Register Integration (Roth)

- “Communicate” results of DDT to main thread
- DDTs are subset of program
  - data-flow corresponds exactly
- Match up instructions at register rename stage
  - if matching PC and physical register inputs -> assign it the same physical register output
- Avoid re-executing instructions already executed
- Early resolution of branch mispredictions
- Also useful for squash reuse
  - speculative code can be viewed as speculative thread
From DDMT to Speculative Slices

• DDMT requires DDTs to be program subsets:
  ○ enables integration

What if we remove that constraint?

• Construct more efficient “DDTs”...
  ○ freedom to optimize
• ...but, requires new DDT mapping mechanism
  ○ no longer a one-to-one correspondence to program
Speculative Slices (Zilles)

- Slices are not allowed to affect architected state
  - Only generate predictions & prefetches
- Removes all correctness constraints from slices:
  - Enables slices to be transformed arbitrarily

- Profile program, identify predictable behaviors
- Transform code to assume these behaviors
  - removes code from slice, improving efficiency
  - results in incorrect computations on uncommon case

Common case efficiency at the expense of occasional mispredicts
Mapping for Speculative Slices

- Need to map predictions to branches in original prog.
- No data-flow correspondence (hence no integration)

Use control-flow

- Prediction generating instruction (in slice) specifies:
  - prediction
  - PC of corresponding branch
  - region of execution for which the prediction is valid
Valid Regions (Zilles)

- Prediction computed assume a particular path (or set of paths)
  - corresponds to a region in the space of all possible executions
- predictions should be destroyed if execution escapes region
- instructions on region boundary are marked
- in practice few markers are needed
Lessons learned from DDMT/Speculative Slices

• Computation can be an efficient means to make predictions
• Can this idea be used in speculative parallelization?
Program Parallelization -- Theme III

- **Traditional view:** master/slave message passing
  - Master divides problem, assigns slaves to pieces
  - Master sends each slave the necessary fraction of data
  - Generally programmer ensures slave’s work is independent
  - Hence, no inter-slave communication

- **A newer view:** master/slave speculative parallelization
  - Master executes “distilled” copy of original program
  - Master forks slaves to execute chunks of original program
  - Master provides start PC and live-in predictions
  - Inter-slave communication to verify live-in predictions
  - Extension of parallel microarchitecture
Master Slave Speculative Parallelization (Zilles)

- Optimizing live-in communication (master/slave)
- Optimizing live-in computation (distilled programs)
- Execution model
Optimizing Live-in Communication

Illustrative Example: loop counter increment

Previous Models:
• Communication Latency Serialized

MSSP:
• Communication Latency Parallelized

Critical Paths
Optimizing Live-in Computation

• In general, computing live-ins not so trivial

• Want to optimize computation of inter-task values

• Tension in previous models

• Single executable:
  ◦ computes live-ins for future tasks (want fast)
  ◦ updates architected state (want correct)

MSSP decomposes problem:

• distilled program (master) allowed to be incorrect
  ◦ enables maximizing performance of the common-case

• original program (slave) allowed to be slow
  ◦ correctly updates architected state, verifies master
Distilled Program Example

- **Average path length reduced by 2/3rds**
- **Significant reduction of static size, taken branches**
- **Correct 99.999% of the time**

Example from *bzip2* (3% of total execution)

Profile-guided optimizations:
- eliminate branches
- inline function
- avoid save/restores
- remove dead code
- register allocate
- reassign logical register
- constant folding

Like traditional optimizations, but not 100% safe
Detailed Execution

Speculative Execution  Parallelized Verification Execution

fork instruction

initiation latency

detection latency
Misspeculation Path

Speculative Execution  Parallelized Verification Execution

misspeculation detected

bad checkpoint

Speculative Multithreading: from Multiscalar to MSSP
Misspeculation Path

Speculative Execution

Parallelized Verification Execution

misspeculation
bad checkpoint
restart latency
misspeculation detected
Misspeculation Path

Speculative Execution Parallelized Verification Execution

e' \{ A, B, C, D \}

misspeculation
critical path
bad checkpoint

r' \{ R, D, I \}

misspeculation detected
Analytical Performance Model

- Model of 3 parameters:
  - $\alpha = \text{speedup of distilled program relative to original}$
  - $P = \text{fraction of correct checkpoints (prediction accuracy)}$
  - $O = \text{normalized overhead} = (I+D+R)/E$

Performance is super-linear with checkpoint accuracy

At high checkpoint accuracy, performance tracks distilled program and is insensitive to inter-core latency
So What?

- Can distilled programs be automatically generated to be fast and accurate?
- We think so.
- Currently developing automatic distiller:
  - early results: not all transformations implemented yet

Implemented transformations achieve results comparable to example from bzip2 (15-40% vs. 22%)
Performance vs. Accuracy

- A continuum of distilled programs exists
  - can turn on/off transformations, set accuracy thresholds

- Curve fit best configurations
  - most benefit achieved with little accuracy impact
  - incremental benefit from trading off accuracy
**MSSP Summary**

- **Master**: executes distilled program, which forks slave threads and predicts their live-in values.
- **Slaves**: perform parallelized execution of original program, verify live-in predictions.

**Model conforms to real world constraints:**
- supports legacy code *(no necessary compiler mod’s)*
  distilled program can be derived from original program
- no verification of program distiller necessary
  distilled program has no correctness constraints
- tolerant of wire latency
  only exposed on rare misspeculations by master
Conclusions

• A variety of different speculative multithreading models of the past decade

• Multiscalar
  ◦ Use speculation to parallelize program execution

• DDMT/Speculative Slices
  ◦ Use speculation to execute critical computations early

• Master/Slave Speculative Parallelization (MSSP)
  ◦ Fusion of Multiscalar/DDMT/Speculative Slices