

# Compile-Time Speculative Scheduling

Wen-mei Hwu

Dept. of ECE

University of Illinois  
at Urbana-Champaign

# Control Speculation

- Executing an instruction before knowing that its execution is required
- Moving an instruction before a decision point for its execution
- Must ensure that execution result unaffected by such movement

# Data Speculation

- Executing an instr. before knowing it can be executed correctly
- Moving a ld and its uses above a st that might access same location
- Must correct the ld and the uses thus moved in case of conflict
- Data speculation is covered in a different section

# Control Speculation

- Moving an instruction above a branch
- Removes control dependences to increase ILP
- Win when branch directions predicted correctly

# Control Speculation

- Approach
  - Compiler optimizes and schedules along predicted path of control
  - Trace scheduling [Fisher] [Ellis]
  - Superblock optimization and scheduling [Impact]
  - Hyperblock optimization and scheduling [Impact]

# Control Speculation Models

- Scheduling error: An ordering of instructions that will cause early program termination or produce results that differ from those of the unscheduled program.
  - Live value must be properly preserved
  - Exception condition must be correctly reported

# Control Spec. Strategies

- Avoid errors
  - do not speculate any instruction that can cause an error.
- Ignore errors
  - assumes likelihood of a real error is small.
- Resolve errors
  - speculates instructions that can cause errors but has some hardware mechanism to resolve the error.

# Safe Speculation

- Compiler analysis to eliminate unnecessary control dependences.
- Instructions that are *always* safe.
- Speculation that will not introduce a new exception.

# Safe Speculation Cont.

- Complex analysis examples (fig.):
  - Branches to ensure legal input operands
  - Loop analysis

# Trivial Safe Analysis

- Structure references and array references with constant indices
- Divide and remainder with non-zero divisor

# Trivial Safe Analysis Cont.

- Memory references inserted by the compiler:
  - to pass parameters between functions
  - to handle spill code
  - to transfer between integer and FP register files [HPPA, SPARC, MIPS]
  - to load floating point constants [HPPA]

# Complex Safe Analysis

- Guarding Branch:
  - Search for an earlier branch that guarantees that the input operands are valid for an instruction.
  - This branch is the farthest point that the instruction may be speculated along that path.

# Complex Safe Analysis

- Loop Analysis:
  - Focuses on array or buffer loads.
  - Using program analysis, determine that the range of locations referenced within the array or buffer are within its dimensions.
  - Loads that meet this condition require no control dependences within the loop.

# General Speculation

- Also called General Code Percolation
- Architecture provides silent versions of instructions that may potentially cause exceptions.
  - Multiflow - silent FP instructions
  - HPPA - silent FP instructions, non-trapping dereferenced null pointer
  - SPARC V9 - silent load instruction

# General Speculation (cont.)

- To move an instr. above a branch, convert it into its silent version.
  - Instrs. thus moved can't cause an exception.
- May miss program error and add to tlb misses and page faults
- No silent stores or I/O references.
- Both Multiflow TRACE and Cydrome Cydra-5 used similar ideas.

# Silent Instructions

- Architecture Support:
  - If a segmentation fault condition occurs to a silent memory load instruction, the instruction is canceled before it reaches the memory system. An arbitrary garbage value is returned.

# Silent Instructions (cont.)

- If a page fault happens to a silent memory load instruction without segmentation fault, the OS page fault handler is immediately invoked as usual. The memory access instruction will be retried after handling the page fault. Extra page faults may occur from code percolation.

# Silent Instructions (cont.)

- If a trap condition occurs to a silent arithmetic instruction, an arbitrary garbage value is deposited into the destination register.
- In all cases, the exception condition that occurred to a silent instruction is either immediately handled or simply ignored.

# Debugging Implications

- If the branch direction differs from compile-time prediction:
  - the garbage value generated by a speculated silent instruction would not be used.
  - the exception condition is correctly ignored since the silent instruction should not have been executed.

# Debugging Implications

- If the branch agrees with compile-time prediction:
  - the exception condition that occurred to a silent instruction is incorrectly ignored.
  - the garbage value generated may be used by a subsequent instruction without warning.
  - not acceptable if exceptions must be reported timely and accurately

# Performance Issues

- Additional page faults
  - Page faults caused by silent loads must be handled right away
    - no support to defer page fault until execution of instruction is confirmed.
  - Additional page faults may result from speculative execution.

# Performance Issues (cont.)

- The number additional page faults should be very small for systems that are designed not to page.
- Similar issues exist if TLB misses are handled through exception mechanism.

# Sentinel Scheduling

- Design Objective
  - Correctly ignore exceptions generated by speculative instructions whose execution turns out to be unnecessary.
  - Correctly report exceptions generated by speculative instructions whose execution is confirmed.

# Sentinel Scheduling

- Design Objectives (Cont.)
  - Support recovery from exceptions thus reported.
  - Provide the option to handle page faults after the need for executing a speculative instruction is confirmed.
  - Minimize the extra hardware and instructions needed to achieve the objectives above.

# Accurate Exception Report

- Each instruction has 2 parts:
  - Non-excepting part which performs the actual operation
  - Sentinel part that flags an exception if necessary
  - Non-excepting part of I can be speculatively executed provided the sentinel part stays in I's home block

# Accurate Exception Report

- The sentinel part of I can be eliminated if there is another instruction in I's home block which uses the result of I OR I is non-excepting and is not the last direct or indirect use of an excepting instruction's destination

# Accurate Exception Report

- Unprotected instruction - an instruction whose sentinel cannot be eliminated.
- If an unprotected instruction is speculatively executed, an explicit instruction must be created to serve as the sentinel

# Architectural Support

- Additional opcodes to specify speculative instruction.
  - by adding speculative version of all opcodes that should be considered for speculative scheduling and that can directly or indirectly cause exceptions.

# Architectural Support (cont.)

- Exception bit (vector) added to each register to mark exceptions caused by a speculative instruction.
  - These bits need to be preserved across context switches.
  - aggregated into a special processor register in Itanium

# Execution Model

- Speculative instructions
  - $\text{src}(I).\text{except} = 0$ 
    - I does not cause an exception
      - normal execution
    - I causes an exception
      - $\text{dest}(I).\text{except} = 1$
      - $\text{dest}(I).\text{data} = \text{pc of } I$

# Execution Model

- Speculative instructions (cont.)
  - $\text{src}(I).\text{except} = 1$ 
    - (exception propagation)
    - $\text{dest}(I).\text{except} = 1,$
    - $\text{dest}(I).\text{data} = \text{src}(I).\text{data}$

# Execution Model

- Non-speculative instructions
  - $\text{src}(I).\text{except} = 0$ 
    - I does not cause an exception - normal execution
    - I causes an exception - I reported as source of exception

# Execution Model

- Non-speculative instructions
  - $\text{src}(I).\text{except} = 1$ 
    - (report exception for speculative instruction)
    - signal exception
    - $\text{src}(I).\text{data}$  is PC of exception

# Scheduling Algorithm

- Identify unprotected instructions
- Perform conventional scheduling
  - if an unprotected instruction is moved above a branch, an explicit sentinel instruction is inserted into list of to-be-scheduled instructions

# Scheduling Algorithm (cont.)

- Explicit sentinel restricted to remain in *I*'s home block with control dependences
- All instructions moved above a branch are marked as speculative

# Recovery from Exception

- Important to allow accurate handling of page faults and TLB misses.
- Issues:
  - ensure that instructions can be retried after the exception condition is handled
  - minimize the negative performance impact in terms of register pressure and instruction count due to recovery.

# Simple Inline Recovery

- Re-execute all instructions that follow the violating instruction.
  - In order to ensure that E can be retried, r1 must be preserved until G finishes checking r5.
    - However, under normal register allocation model, r1 is dead after E.
    - Compiler introduced register reuse
  - See R-tag in ISCA-25 (1998) paper

# Recovery Block

- Copy speculative instructions into recovery blocks
  - One entrance point per potential exception to be reported by a sentinel
  - Code Expansion vs. Efficiency
- Source registers of the instructions not in the recovery blocks are not preserved.
- Instructions re-executed during recovery are reduced.

# Recovery block example

- Recovery block for B

B:  $r1 = \text{MEM}(r2+0)$

E:  $R5 = r1+1$

- Recovery block for C

C:  $r3 = \text{MEM}(r2+4)$

D:  $r4 = r3+1$

F:  $\text{MEM}(r2+4) = r4$

# Multiple Exceptions

- Different basic blocks
  - first sequential exception always reported since check instruction guaranteed to remain in home block of each potential trap-causing instruction
- Same basic block
  - An exception will be signaled but no guarantee it will be the first according to original source code