Presentation 5

Advanced Techniques for Avoiding Misprediction

ההצלחה של ביצוע ספקולטיבי הביאה למצב בו כל misprediction יקר מאוד וגורר תהליך התאוששות מורכבת אחרי ביצוע מספר רב של פקודות לא נכונות. שתי טכניקות מתקדמות למניעת misprediction הן trace cache ו- predication. השיטה הראשונה שומרת סדרות ארוכות של פקודות שהתבצעות בעבר, כולל פקודות סיעוף. השיטה השנייה מחליפה פקודות סיעוף עם ביצוע מותנה, שיטה עדיפה עבור סיעופים שקשה לחזות.

Trace Caching

Slide 2 **Branch Mispredictions Limit ILP**

In order to achieve high performance, superscalar CPUs require a high degree of instruction level parallelism (ILP). Speculative execution, including branch prediction and cache prefetch, is a key technique in providing a large number of independent instructions for parallel execution. With fast decoding, long speculative instruction sequences fill the cache and are moved to the ROB for out-of-order execution.

This means that a **misprediction** causes a very **long recovery process**, involving the cancellation of a large block of executed instructions, the eviction and replacement of many instruction cache lines, and the **decoding and rescheduling** of many instructions in the ROB. Instead of smoothly advancing the **predict-prefetch-decode pipeline**, the CPU enters a recovery mode during which the EUs starve (stall) with no available work. This is called a predict-prefetch-decode bottleneck.

Slide 3 – 4 **Trace Cache**

A **trace** is a long instruction sequence in dynamic execution order, the order of execution determined by the ROB (not the program listing order). A trace may include multiple branch actions (taken/not-taken). Trace cache stores instruction traces constructed dynamically on first run of the instructions in each trace.

With trace cache, branch prediction can be implemented as trace prediction for rapid recovery from a misprediction. If a predicted trace evaluates as a misprediction and the correct prediction is a different trace in cache, then recovery simply involves copying the cached trace to the ROB. This process skips repetition of the slower process of instruction fetch, branch prediction, decoding, and dynamic rescheduling.

The trace cache is **constructed** on the first run of a set of instructions. These instructions undergo the usual pipeline stages: loading of cache lines, instruction fetch, branch prediction, decoding instructions into the instruction pool, and scheduling the instructions for dynamic execution in EUs. A **trace cache entry** is started on the target or fall-through of a branch instruction. The trace embeds the next few branch actions, with the instructions in validated taken and not-taken execution paths. The entry ID is the PC of the first instruction and a flag representing the sequence of taken and not-taken actions embedded in the trace.

Slide 5 – 8 **Trace Cache**

Slide 5 shows a program fragment consisting of 8 program blocks (PB0 to PB7) and 4 branch instructions — two while loops (based on conditions C2 and C4) and an if-else control structure (one based on conditions C1 and one unconditional).

The red arrow shows the behavior of the conditional branch at the end of the **while(C4) loop**, and the purple arrow shows the behavior of the **while(C2) loop**. The blue arrow shows the behavior of the **if(C1)** structure. The green arrow shows the program skipping IB5 (unconditionally) when condition C1 is false.

Slide 6 shows that the program has 32 instructions — either branch instructions or members of instructions blocks. The runtime sequence with particular values of the conditions C1 – C4 is shown on slides 7 and 8. First IB0 and IB1 complete. Because **if(C1 = 0)** is false, the program continues with IB2 and IB3. IB3 runs 4 times in a loop, followed by IB4 and IB6, then back to IB1.

The trace cache is assembled beginning with the fall-through of the first branch, which is IB2.

Trace T1 contains IB2, IB3, J(C2).

Trace T2 contains IB3, IB3, J(C2).

Trace T3 contains IB3, IB4, J(C3=1).

Trace T4 contains IB6, J(C4), IB1, J(C1).

The correct loop sequence is T1 T2 T3 T4.

The branch at the end of a trace is predicted, pointing to a new trace. The trace is loaded and validated while running.

At the end of T1, the taken prediction of J(C2) points to T2. (valid)

At the end of T2, the taken prediction of J(C2) points to T2. (misprediction)

T2 is reloaded, but this is mispredicted — IB3 would run 5 times instead of 4.

The ROB cancels the instructions of T2, and the valid trace T3 replaces T2.

At the end of T3, the unconditional branch points to T4. (valid)

At the end of T4, the not taken prediction of J(C1) points to T2 (valid).

At the end of T2, the updated not taken prediction of J(C2) points to T3. (valid)

Slide 9 – 11 Trace Cache Header

A trace cache entry consists of an entry header and the instructions in the trace. The header is shown on slide 9. The fields are:

Start PC to identify the first instruction of the trace.

Flag indicating the sequence of branch actions in the trace.

Mask to show which flag positions hold valid information.

End type to indicate the type of the last instruction in the trace, conditional or unconditional branch, or not a branch (fall-through is next PC by default).

Fall-through / target provides the PC for next instruction fetch.

Trace headers for the example are shown on slide 10.

In order to make cache updates efficient, the length of a trace cache entry is set to the length of a cache line. The number of branches that can be stored in a trace is limited by the speed of the branch predictor. A trace is always terminated (filled with NOPs) at a subroutine return, because the return address is difficult to predict.

Slide 12 – 15 Instruction Fetch Unit

Slide 12 shows the relationship of **trace cache** to the **instruction fetch logic** and **branch prediction**. The next-PC logic predicts individual branches and the **multiple branch predictor** (MBP) predicts the multiple branches within traces. A new PC address is sent to instruction cache and the MBP in parallel. If the PC misses in the trace cache (no trace is found starting at that address), then the instruction is fetched from instruction cache, and the MBP updates the PC register. If the PC hits in trace cache, then the access to instruction cache is cancelled and a trace is fetched to the instruction pool. The MBP must pick the trace by predicting the multiple branches stored in multiple trace entries starting at the same PC.

The single branch predictor (**next-PC logic**) is a one-level or two-level branch prediction system of the type discussed in presentation 4. The **multiple branch predictor** predicts m branches that are embedded in a trace. The prediction is an m-bit bitmap — each bit is set as the last prediction made for that branch in the single branch predictor. When the single branch predictor makes a PC prediction that hits in trace cache, the MBP reads the bitmap for the branches embedded in that trace to predict the taken / not taken sequence, which chooses the particular trace in cache starting with that PC.

The BHR and PHB for the trace example is given on slide 14. The MBP is indexed by the PC, which is always a target or fall-through instruction.

The trace cache for the Intel Nehalem architecture is shown on slide 15.

Branch Predication

Slide 17 Another Look at Branches

The application of branch instructions can be divided into two basic contexts with distinct characteristics, **loops** and **if-conditions**. These contexts are compared in the table on slide 17.

In a simple loop, a conditional branch appears at the end of a long code sequence and jumps backward in order to reuse the code on a stream of data. The loop can also be built with a conditional forward branch (that breaks from the loop) at the top of the code and an unconditional branch at the end. The branch instructions are a small part of the loop code and are highly predictable. Branch prediction performs well on loops.

In an if-condition, the branch generally jumps forward to navigate among alternative code blocks based on a number of dynamic conditions to be tested. The branch instructions are often a high proportion of the code block, and are difficult to predict. Branch prediction functions poorly with if-condition structures, especially on superscalars that may be executing multiple branches per clock cycle.

Predication is an alternative to prediction.

Slide 18 Predication

An ISA that implements **predication** defines a set of 1-bit predication registers p0,...,pN that can be set by ALU instructions. A predicated instruction is run conditionally, so that the code

```
<p1> Instruction
```

causes Instruction to be executed if **`p1 = 1`**, or Instruction to be cancelled (NOP) if **`p1 = 0`**. This permits the conversion of if-condition branches to conditional execution, eliminating the branch stall at the cost of canceling extra work. For example, the code

```
     BEQZ R1, L1
     SUBI R1, R1, #1
L1:  ADD R2, R2, R1
```

skips the SUBI instruction when R1 is zero. The equivalent predicated code is

```
      p1 = CMP R1, 0
<p1>  SUBI R1, R1, #1
      ADD R2, R2, R1
```

Which cancels the SUBI instruction when R1 is zero. There is no clear advantage to the predicated form on a simple scalar CPU — the advantage is seen on a superscalar with dynamic scheduling.

Slide 19 – 21 Predication in IA-64

IA-64 is Intel's 64-bit ISA for the Itanium series processors. It defines 16 predicate registers, where p0 always equals 1. There are 48 virtual predicate registers for automatic renaming.

The general syntax of an IA-64 instruction is

`[(qp)] mnemonic[.comp1][.comp2] dests = srcs`

where **`qp`** is an optional predicate, **`.comp`** is a comparison, and **`dests`** can be predicate registers.

Slide 20 shows a simple switch control structure in C, with case 'a' distinct from case 'b'. The compilation for IA-32 (x86 ISA) is:

```
        MOV  EAX, [type]          ; copy type to register
        CMP EAX, 'a'              ; compare register to 'a'
        JNE L1                    ; if not equal then skip to next compare
        ADD EAX, 10               ; if equal, add 10 to register
        JMP L2                    ; skip to end
    L1: CMP EAX, 'b'              ; compare register to 'b'
        JNE L2                    ; if not equal then skip to end
        ADD EAX, 20               ; if equal, add 20 to register
    L2: MOV [type], EAX           ; store new value to type
```

The code has 9 instructions with 3 branch instructions that cannot be predicted because they run only one time. The branch prediction is not-taken for all three. There is one misprediction (stall) for either 'a' or 'b' and 2 mispredictions for neither.

The compilation for IA-64 is:

```
    mov  r1 = type                ; copy *type to register
    ld4  r2 = [r1]                ; copy type to register
    cmp.eq  p1 = 'a',r2           ; p1 = result of compare register to 'a'
    cmp.eq  p3 = 'b',r2           ; p3 = result of compare register to 'b'
    (p1)  add r2 = 10,r2          ; if p1 = 1, add 10 to register
    (p3)  add r2 = 20,r2          ; if p3 = 1, add 30 to register
    st4  [r1] = r2                ; store new value to type
```

There are 7 instructions, with no branches and no mispredictions. After scheduling for the Itanium, two instructions are executed in CC3, but one or both will be cancelled. If type is not 'a' or 'b' then no useful work is performed in CC3 — the equivalent of a stall. The IA-64 version removes 1 stall for each outcome.

Slide 23 Predication in ARM

The ARM ISA defines predicated versions of certain instructions that are executed conditionally by testing the status flags. The instruction SUBS performs SUB and sets the flags. The instruction ADDNE performs ADD only if the Zero flag is not set (not equal). The instruction SUBEQ performs SUB only if the Zero flag is set (equal). No branches are required.

Slide 23 Predication in PIC

PIC is a family of microcontrollers widely used in embedded systems. The PIC ISA defines instructions that test conditions and conditionally cancel the next instruction. The assembly code on slide 24 shows an implementation of the switch:

```
switch (PIR1)
{
  case 'TMR1IF':
    T1_INT;
    break;
  case 'ADIF':
    AD_INT;
    break;
  case 'LCDIF':
    LCD_INT;
    break;
  case 'RBIF':
    PORTB_INT;
    break;
  default:
    INT_ERROR_LP1;
    break;
}
```

The assembly code cancels the GOTO (branch) instructions when the condition is not matched:

```
BTFSC PIR1, TMR1IF      ; Cancel next if (PIR1<TMR1IF> == 0)
GOTO T1_INT             ; branch to Timer1 ISR
BTFSC PIR1, ADIF        ; Cancel next if (PIR1<ADIF> == 0)
GOTO AD_INT             ; branch to A/D ISR
BTFSC PIR1, LCDIF       ; Cancel next if (PIR1<LCDIF> == 0)
GOTO LCD_INT            ; branch to LCD ISR
BTFSC PIR1, RBIF        ; Cancel next if (PIR1<RBIF> == 0)
GOTO PORTB_INT          ; branch to PortB ISR
GOTO INT_ERROR_LP1      ; default ISR
```

Slide 24 – 33 Generalized Predication Model

The predication model can be generalized by setting multiple predicate registers per compare instruction. In this syntax,

`p1 = CMP.type1 src1,src2 <p>`	comparison type1 sets register p1
`p1,p2 = CMP.type1.type2 src1,src2 <p>`	comparison type1 sets register p1
	comparison type2 sets register p2

Complex conditions can be encoded into the predicate registers by predicating the comparisons with <p> and by modifying predicates after being set.

The syntax shown on slide 25, where a comparison is notated

```
C = CMP.type1.type2
```

In addition to the usual arithmetic comparisons, the type may include the comparison C (type T) or its logical complement C' (type F). The type also specifies the assignment:

U (unconditional)

The predicate register is always set.

Its value is set to **`(p && C)`** for type UT or to **`(p && C')`** for UF.

C (conditional)

The register is only set if **`p = 1`** — otherwise it is left unchanged.

The register is set to C for CT or C' for CF.

O (OR)

If **`(p && C) = 1`** for OT the register is set to 1.

If **`(p && C') = 1`** for OF the register is set to 1.

Otherwise, it is left unchanged.

A (AND)

If **`(p && !C) = 1`** for OT the register is set to 0.

If **`(p && !C') = 1`** for OF the register is set to 0.

Otherwise, it is left unchanged.

∨

If **`(p || C) = 1`** for OT the register is set to 1.

If **`(p || C') = 1`** for OF the register is set to 1.

Otherwise, it is left unchanged.

∧

If **`(p && C) = 1`** for OT the register is set to 1.

If **`(p && C') = 1`** for OF the register is set to 1.

Otherwise, it is left unchanged.

Additional instructions convenient for manipulating predicate registers are shown on slide 26.

Slide 27 shows an example in which a sequence of 5 instructions is used to set predicate registers p1 and p2. The first instruction sets p1 to the value of comparison C1. The second instruction overwrites p1 to 1 on condition C2. The third instruction overwrites p1 to 0 on condition C3. The fourth instruction sets p2 to 1. The fifth instruction overwrites p2 to 0 if either condition C4 is 0 or p1 is 0.

Slide 28 shows a short while loop in C and its compilation for IA-32. The gcc compiler uses two branch instructions — the first runs once (mispredicted) and the second runs on every iteration (mispredicted on the first and last iteration). The predicated version unconditionally sets p1 to the loop condition and cancels the loop instructions when it is no longer satisfied. It uses one branch that runs on each iteration and misses once (it is cancelled on the last iteration). In dynamic scheduling with speculative execution, the pointless execution of the cancelled instructions can be less costly than a misprediction.

Slides 29 and 30 show two similar while loops with one additional condition. The compilation uses an additional branch to implement the compound condition. The predicated version implements the compound condition by unconditionally setting p1 to the first condition and then conditionally overwriting p1 with the second condition.

Slide 31 shows a similar while loops with 3 conditions — the compilation uses 4 branch instructions to implement the compound condition. The predicated version implements the compound condition by unconditionally setting p1 to the first condition and then conditionally overwriting p1 with the second and third conditions.

Slide 32 shows a while loop with an if-condition in each iteration. The IA-32 compilation requires 5 branch instructions:

```
        mov [x], 0x0        ; x ← 0
        jmp L3              ; start at comparison
L1:     mov eax, [x]        ; eax ← x
        mov [y], eax        ; y ← eax = x
        add [x], 0x1        ; x++
        mov eax, [x]        ; eax ← x
        and eax, 0x1        ; zero all but low order bit of eax
        test eax, eax       ; compare eax with zero
        jne L2              ; if not zero skip to L2 (skip the if block)
        mov eax, [y]        ; eax ← y
        add eax, [x]        ; eax ← x + eax
        mov [z], eax        ; z ← eax
        jmp L3              ; skip to L3 (skip the else block)
L2:     mov eax, [x]        ; eax ← x
        sub eax, [y]        ; eax ← eax - y
L3:     cmp [x], 0x9        ; compare x < 9
        ja end              ; if x > 9 jump to end
        cmp [y], 0x4        ; compare y > 4
        jna L1              ; loop if y < 5
end
```

Slide 33 shows the predicated implementation. The loop is again controlled by setting a predication register p1 with a condition and conditionally overwriting it after a second

comparison. The if-condition is handled with two new predicate registers p2 and p3, which are set by the true or false result of comparing register EAX with zero. Either the instructions predicated with p3 or the instructions predicated with p3 will be cancelled.

Slide 34 – 35 Hyperblock Formation

The **conversion** of an if-else control block from branching to predication is shown on slide 34. The typical if-else block begins with a **Head** that ends with a conditional branch to **if-block B**. Below the Head is **else-block A**, which ends with a jump to skip over the **if-block B** to the **Tail**.

The first step in conversion is to convert the conditional branch to a predicate set as p and p', and to predicate the jump to B with p and the block A with p'.

The second step predicates B with p, and copies the Tail, one predicated with p and one with p'.

The last step is to reorder the blocks.

> If p = 1, the jump to B, B, and the Tail for B will run. A and its copy of Tail will be cancelled.
>
> If p = 0, the jump is cancelled, along with B and its copy of Tail. A and its copy of the Tail will run.

The program now has only one absolute branch, which is only mispredicted on its first run. In a superscalar with trace cache, both A+Tail' and B+Tail are in trace cache after 1 run, so that speculative execution can run with high efficiency and minimum misprediction recovery.

Hyperblock conversion can be performed as a compiler optimization for an ISA that defines predication. Since the x86 does not permit predication, it is not found in the PC market.

Slide 36 – 37 Multipath Execution

In multipath execution, the block predicated by p and the block predicated by p' can run in parallel. Both blocks can even be scheduled speculatively before p is evaluated. When the value of p is found, one block is cancelled in the ROB and the other block can be retired and committed to state. A simple example is shown on slide 36.

The table at the top of slide 37 shows an execution example on a superscalar DLX. If R3 is zero I_1 runs as fall-through with no stall, followed by J I_3 that has a 1 CC stall. If R3 is not zero, then BNEZ executes with a 1 CC stall jumping to I_2.

The table at the bottom of slide 37 shows the same CPU with predication. All 4 instructions can be run in parallel. When the result of the first instruction is found, either I_1 or I_2 is cancelled in the ROB.