Module-3
CSEN 3104
Lecture 28
26/09/2019

Dr. Debranjan Sarkar

Branch Prediction

Dependence Analysis

- To execute several program segments in parallel, each segment is to be independent of the other segments
- 3 types of dependences:
  - Data dependence (The ordering relationships between statements)
  - Control dependence (Order of execution of statements cannot be determined before run time)
  - Resource dependence (refers to the conflicts in using shared resources, such as integer units, floating-point units, registers etc.)
- Dependence analysis determines whether it is safe to reorder or parallelize statements

Control Dependency

- Control dependency is a situation in which a program instruction executes if the previous instruction evaluates in a way that allows its execution
- In the following example, the statement S2 runs only if the predicate in S1 is false

```
S1 if x > 2 go to L1
S2 y = 3
S3 L1:    z = y + 1
```

- The statement S2 is said to be control dependent on S1 (written as S1 δ^c S2) as S2's execution is conditionally guarded by S1

Control Dependence in a loop

- Loop Independent Dependence
- A statement in one iteration of a loop depends only on a statement in the same iteration of the loop

```
    for (i = 0; i < 4; i++)
    {
S1:    b[i] = 8;
S2:    a[i] = b[i] + 10;
    }
```

- Loop independent dependence between statements S1 and S2 in the same iteration

Control Dependence in a loop

- Loop-dependent Dependence
- A statement in one iteration of a loop depends on a statement in a different iteration of the same loop

```
    for (i = 0; i < 4; i++)
    {
S1:    b[i] = 8;
S2:    a[i] = b[i-1] + 10;
    }
```

- Loop-dependent dependence exists between i^{th} iteration of statement S2 and

  $(i-1)^{th}$ iteration of S1

- Statement S2 cannot proceed until statement S1 in the previous iteration finishes

Control Dependency

- Conditional branch statements cannot be resolved until run time
- Different paths taken after a conditional branch may introduce or eliminate data dependence among instructions
- Control dependence often prohibits parallelism from being exploited
- Techniques to get around the control dependencies in order to exploit more parallelism:
  - Compiler techniques
  - Hardware branch prediction techniques

Handling Control Dependency

- How to Handle Control Dependences?
- Critical to keep the pipeline full with correct sequence of dynamic instructions.
- Potential solutions if the instruction is a control-flow instruction:
  - Stall the pipeline until we know the next fetch address
  - Guess the next fetch address (branch prediction)
  - Employ delayed branching (branch delay slot)

Branch Prediction

- 3 different kinds of branches:
  - Forward conditional branches (PC is changed to point to an address forward in the instruction stream)
  - Backward conditional branches (PC is changed to point backward in the instruction stream)
  - Unconditional branches (e.g. jumps, procedure calls and returns)
- Branch prediction attempts to guess whether a conditional jump will be taken or not
- 2 Branch prediction schemes
  - Static
  - Dynamic

Static Branch Prediction

- Based on the information that was gathered before the execution of the program
- Predicts always the same direction for the same branch during the whole program execution
- 2 types
  - Hardware-fixed prediction
  - Compiler-directed prediction
- Hardware-fixed direction mechanisms can be:
  - Predict always not taken
  - Predict always taken
  - For ‘Backward Branch’ predict “Taken”, for ‘Forward Branch’ predict “Not Taken”
- Sometimes a bit in the branch opcode allows the compiler to decide the prediction direction
- All decisions are made at compile time, not at run time

Using Branch statistics for static prediction

- Nearly 60% of the forward conditional branches are taken
- About 85% of the backward conditional branches are taken (loops)
- So, a Static Predictor can just look at the offset (distance forward or backward from current PC) for conditional branches as soon as the instruction is decoded
- Backward branches will be predicted to be taken, since that is the most common case
- The accuracy of the static predictor will depend on the type of code being executed, as well as the coding style used by the programmer

Dynamic Branch Prediction

- It is based on recent branch history to predict whether or not the branch would be taken next time when it occurs
- During the start-up phase of the program execution
  - a static branch prediction might be effective
  - the history information is gathered
- When history information is gathered, dynamic branch prediction gets effective
- In general, dynamic branch prediction gives better results than static branch prediction, but at the cost of increased hardware complexity

Dynamic Branch Prediction (1-bit)

- Example

```
for (i = 0; i < 5; i++)
{
a[i] =a[i] + 5;
}
```

- We would predict whether branch is to be taken or not
- In 1-bit branch prediction method, there are 2 states (show figure)
  - 0 -> Not likely to be taken state. If in ‘0’ state, predict that branch is not to be taken (NT)
  - 1 -> Likely to be taken state. If in ‘1’ state, predict that branch is to be taken (T)
- Let us assume that initial state is ‘1’
- So initial prediction is “Branch is to be taken (T)”

Dynamic Branch Prediction (1-bit)

- Value of i 0 1 2 3 4 5 6 7 8
- Present state 1 1 1 1 1 1 0 1 1
- Prediction T T T T T T NT T T
- Actual T T T T T NT T T T
- Next state 1 1 1 1 1 0 1 1 1
- Right / wrong √ √ √ √ √ X X √ √

- 2 mis-predictions in case of anomaly
- To reduce the number of mis-predictions, we use 2-bit branch prediction

Dynamic Branch Prediction (2-bit)

- In 2-bit branch prediction method, there are 4 states (show figure)
  - 00 -> strongly not likely to be taken state
  - 01 -> weakly not likely to be taken state
  - 10 -> weakly likely to be taken state
  - 11 -> strongly likely to be taken state
- MSB indicates whether branch is likely
  - to be taken (MSB = 1), or
  - not to be taken (MSB = 0)
- LSB indicates whether the prediction is
  - strong (LSB = 0), or
  - weak (LSB =1)

Dynamic Branch Prediction (2-bit)

- Let us assume that initial state is ‘10’
- So initial prediction is “Branch is to be taken” and the prediction is correct
- Value of i 0 1 2 3 4 5 6 7
- Present state 10 11 11 11 11 11 10 11
- Prediction T T T T T T T T
- Actual T T T T T NT T T
- Next state 11 11 11 11 11 10 11 11
- Right / wrong √ √ √ √ √ X √ √

- Only 1 mis-prediction in case of anomaly

Branch History Buffer (BHB)

- It is a table with three (3) columns (show figure)
  - Branch Instruction Address
  - Branch Target Address
  - Branch Prediction statistics
- If the Branch Prediction Statistics indicates that the branch is likely to be taken, in the 2nd cycle, the instruction from the target address is fetched.
- If the prediction is not correct, the prediction statistics is changed accrordingly

Thank you

Module-3
CSEN 3104
Lecture 29
15/10/2019

Dr. Debranjan Sarkar

Delayed Branch

Delayed Branch

- The compiler detects the branch instruction
- Rearranges the machine language code sequence by inserting
  - useful instructions, or
  - NOPs (No operations)
- This keeps the pipeline operating without interruptions

Delayed Branch

- Assume branch delay of one cycle
- If branch taken, execution is:
  - Branch instruction
  - Branch delay instruction
  - Branch target
- If branch not taken, execution is:
  - Branch instruction
  - Branch delay instruction
  - Branch Instruction + 2

```
          BEQZ R1, L1
          branch delay
instruction
          instruction + 2
          instruction + 3
L1:       branch target
          branch target + 1
          branch target + 2
```

- Instruction immediately following branch is executed irrespective of whether the branch is taken or not
- Rely on compiler to make successor instructions valid and useful

Example of Delayed Branch

- I1: Load R1, A
- I2: Decrement R3, 1
- I3: Branch zero R3, I5
- I4: Add R2, R4
- I5: Subtract R5, R6
- I6: Store R5, B

Original Program

- I2: Decrement R3, 1
- I3: Branch zero R3, I5
- I1: Load R1, A
- I4: Add R2, R4
- I5: Subtract R5, R6
- I6: Store R5, B

Reordered Instructions

Scheduling branch delay slots

- Where to get instructions to fill branch delay slot? (Show figures)

  (A) From before branch instruction

  (B) From the target address: only valuable when branch taken

  (C) From fall through: only valuable when branch not taken

Scheduling branch delay slots

- If taken from before branch
  - branch must not depend on rescheduled instruction
  - always improves performance
- If taken from branch target
  - must be OK to execute rescheduled instructions if branch not taken
  - may need to duplicate instructions
  - performance improved when branch taken
- If taken from fall through
  - must be OK to execute instructions if branch taken
  - improves performance when branch not taken

Assignment

- Consider the following program: (assume opcode <src>, <dest> format):

```
    Add R3, R2
    Sub R3, R4
    Add R2, R1
    Mov R1, [R4]   ; write to memory location pointed to by R4
    Jnz R1, ThisPlace
    ...  ...  ...  ...  ...
    ...  ...  ...  ...  ...
ThisPlace: <some code>
```

- Assuming a delay slot value of 3, rewrite the code to exploit the delayed branching mechanism.

Problem

- Assume that branches comprise 20% of all instructions. Also assume that the branch prediction is 80% accurate and incurs a 2 cycle stall on each mis-prediction. What is the impact of control hazards on the CPI of the pipelined processor? Ignore all other sources of pipeline hazards
- Solution:

  CPI without control hazards = 1

  Added CPI due to control hazards

  = Branch frequency * (1 – Branch prediction accuracy) * Stall penalty = 20% * (1 - 80%) * 2 = 0.08

  CPI with control hazards = 1 + 0.08 = 1.08

Speedup Performance Law (Amdahl's law)

- In many applications requiring a real-time response, the computational workload is often fixed with a fixed problem size
- As the number of processors increases in a parallel computer, the fixed load is distributed to more processors for parallel execution
- The main objective is to produce the results as soon as possible (i.e. minimum turn-around time)
- Speedup obtained for time-critical applications is called fixed-load speedup
- Amdahl's law is based on a fixed workload (or problem size)
- Speedup of n processor system is defined using a ratio of execution time:

  $S_n = T_1 / T_n$ (where T_1 is the time taken by a single processor and T_n is the time taken by a system with n number of processors)

Speedup Performance Law (Amdahl's law)

- If α is the proportion of a program (with workload W) that remains serial and cannot be made parallel, and (1-α) is the proportion that can be made parallel, then S_n can be written as:

$$S_n = (W/1)/((\alpha W)/1 + ((1- \alpha)\ W)/n) = n\ /\ (1 + (n-1)\ \alpha)$$

- This is known as Amdahl's law
- Amdahl's law may be restated as follows:
- In parallelization, if P is the proportion of a program that can be made parallel, and (1-P) is the proportion that remains serial, then the speedup that can be achieved using N number of processors is 1/((1-P)+(P/N))

Speedup Performance Law (Amdahl's law)

- Amdahl's law assumes that the system is used either in a pure sequential mode on one processor or in a fully parallel mode using N processors
- In Amdahl's law, computational workload W is fixed while the number of processors that can work on W can be increased
- If N tends to infinity then the maximum speedup tends to 1/(1-P) or 1/α
- This means the best speedup one can expect is upper-bounded by 1/(1-P) or 1/α, regardless of how many processors are employed
- Notice that the speedup can NOT be increased to infinity even if the number of processors is increased to infinity

Speedup Performance Law (Amdahl's law)

- Speedup is limited by the total time needed for the sequential (serial) part of the program. For 10 hours of computing, if we can parallelize 9 hours of computing and 1 hour cannot be parallelized, then our maximum speedup is limited to 10X
- Show the graph showing speedup vs number of processors for different values of P (= 1 – α)
- This shows that the system performance cannot be high as long as the serial fraction α exists
- This α is called the sequential bottleneck in a program
- The problem of sequential bottleneck cannot be solved just by increasing the number of processors in the system. The real problem lies in the existence of a sequential fraction of the code

Example of Amdahl's Law

- A program needs 20 hours using a single processor core
- A particular part of the program which takes 1 hour to execute cannot be parallelized
- While the remaining 19 hours of execution time can be parallelized

- Solution:
- Regardless of how many processors are devoted to a parallelized execution of this program, the minimum execution time cannot be less than that critical one hour
- Here, P = 19/20 = 0.95
- Hence, the theoretical speedup is limited to 1/(1-P) = 1/0.05 = 20 i.e. at most 20 times
- For this reason, parallel computing with many processors is useful only for highly parallelizable programs

Gustafson’s Law

- Amdahl's law applies only to the cases where the problem size is fixed.
- That means the workload does not change with respect to the improvement of the resources
- Gustafson's law proposes that programmers tend to set the size of problems to fully exploit the computing power that becomes available as the resources improve
- Therefore, if faster equipment is available, larger problems can be solved within the same time
- Gustafson's law gives the theoretical speedup of the execution of a task *at fixed execution time* that can be expected of a system whose resources are improved

Gustafson's Law

- Suppose you have an application taking a time t_P to be executed on N processing units. Of that computing time, a fraction (1-P) must be run sequentially. Accordingly, this application would run on a fully sequential machine in a time t_S equal to
- $t_s = (1-P)\ t_P + NPt_P$
- If we increase the problem size, we can increase the number of processing units to keep the fraction of time the code is executed in parallel equal to $P.t_P$. In this case, the sequential execution time increases with N which now becomes a measure of the problem size. The speedup then becomes
- $S = ((1-P)\ t_P + NP\ t_P) / t_P = (1-P) + NP$
- The efficiency would then be
- $E = S/N = (1-P)/N + P$
- The efficiency tends to P for increasing N

Amdahl’s Law versus Gustafson’s Law

- We are looking at the same problem from different perspectives. Amdahl's law says that if you have, say, 100 more CPUs, how much faster can you solve the same problem?
- Gustafson's law is saying, if a parallel computer with 100 CPUs can solve this problem in 30 minutes, how long would it take for a computer with just ONE such CPU to solve the same problem?

Amdahl’s Law versus Gustafson’s Law

- **Amdahl's law**
- Fix execution time on a single processor
- $s + p$ = serial part + parallelizable part = 1 (normalized serial time)
- Assume problem fits in memory of serial computer
- Fixed-size speedup $S_{fixed\_size} = (s + p)/(s + p/n) = 1/(s+(1-s)/n)$
- **Gustafson's law**
- Fix execution time on a parallel computer (multiple processors)
- $s + p$ = serial part + parallelizable part = 1 (normalized parallel time)
- $s + np$ = execution time on a single processor
- Assume problem fits in memory of parallel computer
- Scaled Speedup $S_{Scaled} = (s + np)/(s + p) = n + (1-n)\, s$

Thank you