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