# P6 Programming Tips

## Three Tips for P6 Code

- In general the areas for code optimization are potentially numerous
- The greatest improvement will be seen by paying attention to three specific areas
  - Branch Prediction
  - Partial Stall Removal
  - Data Alignment

The following slides define three key areas of programming for P6 performance, and will provide details on addressing them.

## Branch Prediction Improvements

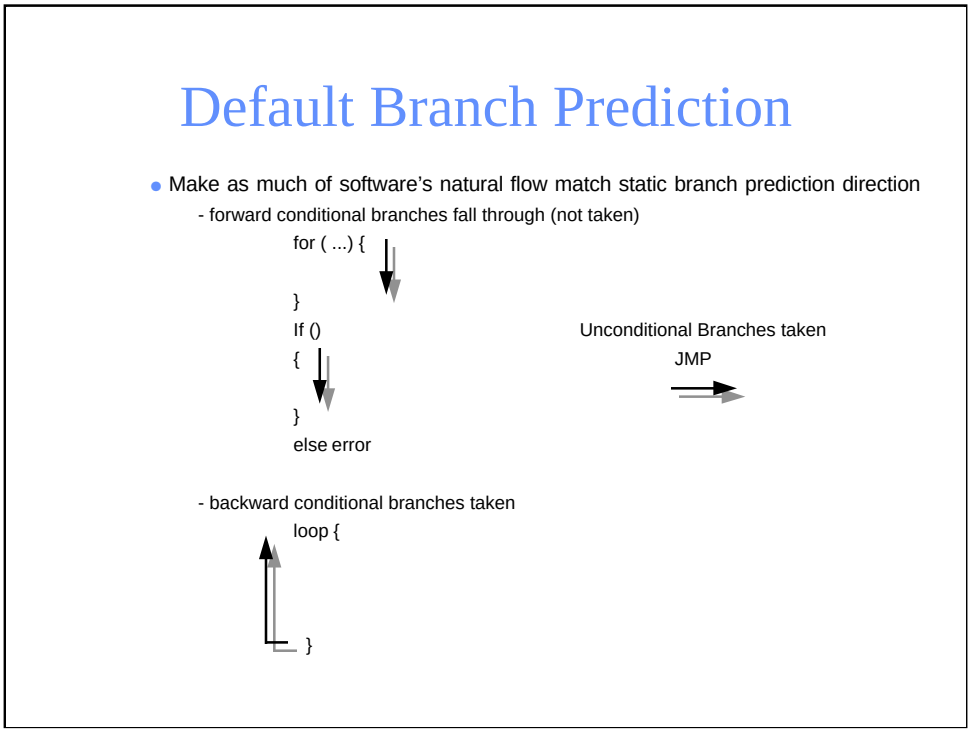
- Improving Branch Prediction is one of the most important optimizations
  - Reduce the number of branches
  - Follow static prediction algorithm
  - Ensure that each CALL has a matching RETurn
  - Identify and Reduce Unpredictable Branches
    - Move Frequent Cases in Switch to If's
  - Don't intermingle Data and Instructions
  - Unroll very short loops

The mixing of data in the instruction stream causes the front end of the machine to be sent off on a fruitless mission. Decoder bandwidth is consumed and in some cases data may be speculatively executed. This is not damaging in the sense that the external world is aware of this but it is damaging in the sense that BTB values may be corrupted and ICache locations are polluted. There is also potential for CALL /RETurn synchronization to be lost unnecessarily.

The flow diagram of static prediction demonstrates how decisions will be made up to the Decoder stage. Matching this flow in software will enable the most optimum execution speed. The example shown highlights the concept of having only error conditions represented as forward conditional branches.

i.e. IF (ERROR) Panic(); equates to:

```
MOV     EAX, ERROR
CMP     EAX,TRUE
JE      PANIC
Continuation Code
        "
        "
        "
Panic:  Error Code
```




The static prediction algorithm is depicted in the flow diagram above. It is best exampled by following the decision tree down the center. That is, if there has been **no prediction** from the BTB **and** the branch is **relative** to the current EIP **and** the branch is **conditional and** the sense of the branch is **backwards** then it will be predicted to be taken by the static algorithm. The BTB will be updated so that the 5-6 clocks expended on this pass through the processor will be reduced to one the next time this branch is seen.

## Eliminate Partial Stalls

- **Reading a register in 32 bit form after writing it in 8/16 bit form causes execution to stall***

```
MOV  EAX, 0          MOV  EAX, 0
MOV  AX, 10          MOV  AL, 17
ADD  EAX, 5          INC  EAX
```

Note: P6 is able to track upper registers if XOR or SUB are used. For example, if

```
XOR  EAX, EAX  or
SUB  EAX, EAX
```

were used instead of MOV EAX, O there would be no partial stall.

* stall is a delay in program execution, waiting for an OPCODE to complete execution

Register Renaming is enabled by P6 having numerous internal registers that can be allocated for use when a general purpose register is written. These registers are RETIRED into the real, general purpose, register file in. This allows the parallelism that exists across multiple basic blocks to be exposed. The flags register itself in not renamed, however each instruction carries with it the flag bits that the instruction will modify when it is retired into the real flags register. There is no renaming of the segment registers.

The implication of register renaming is that writing a register in it's 8/16 bit form which causes it to be renamed and then referencing it in it's 32 bit form cause a "stall". A "stall" is where code execution cannot proceed any further until all preceding instructions have been retired i.e.. updated the real register file. The reason for this is fairly obvious. For example the lower half of the EAX register is addressable as an independent register. The writing of a value to the lower half of the register and then accessing all of the 32 bits of the register assumes that the previous instructions that modified the upper half of the same register (that were also renamed) have completed through to retirement and made the value of the upper half of the registeravailable. Many of the cases that would cause this will be seen and special cased by the Decoder. However this practice should be understood and wherever possible removed.

# P6 Alignment

- As with Intel486™ Processor DATA alignment significantly impacts processor performance, CODE alignment also but to a much lesser degree
  - Align DATA: 16 bit variables on even boundaries
    32 bit variables on 4 byte boundaries
    64 bit variables on 8 byte boundaries
    80 bit variables on 16 byte boundaries
    128 bit variables on 16 byte boundaries
  - Align CODE: Major Code block's, Critical Loop Headers, Interrupt Service Routine's aligned as per the Intel486™ Processor
    (16 byte boundaries)

Data alignment is very important on P6. Misaligned data accesses are completed at retirement and as such are expensive. All data should be aligned on the data types natural boundary.

Code alignment is not so critical. Once the processor is executing in it's speculative mode and the pipeline is filled the small one clock impact caused by a misaligned code reference will be absorbed.

Code alignment is important when the pipeline is empty as in the case of Interrupts or a branch misprediction. It is not always possible to know if a branch will mispredict frequently but if it is suspected then the branch target address should be 16byte aligned.

## Where do I get more data?

Reference the “32-bit Optimization Guide” document included on this CD