CODING GUIDELINES FOR PIPELINED PROCESSORS

James W. Rymarczyk
International Business Machines Corporation
P. O. Box 390
Poughkeepsie, New York 12602

ABSTRACT

This paper is a tutorial for assembly language programmers of pipelined processors. It describes the general characteristics of pipelined processors and presents a collection of coding guidelines for them. These guidelines are particularly significant to compiler developers who determine object code patterns.

INTRODUCTION

This paper presents instruction coding guidelines, most of which have been known for over ten years but have not been widely communicated. These guidelines are particularly significant to compiler developers who determine object code patterns.

Ten years ago, this paper would have been relevant only to programmers of the largest and fastest processors. Today, with much higher levels of digital circuit integration, pipelined processor organizations are being used more widely, even in minicomputers and microcomputers.

A program that complies with these guidelines will execute more efficiently on a pipelined processor than a program that does not. Moreover, these guidelines do not degrade performance on nonpipelined processors, except where noted.

Adherence to these guidelines may, in some instances, reduce the clarity of programs and require the use of additional comments. For example, some of the guidelines call for increasing the separation of instructions that are logically related. This effect is inconsequential in most cases, especially for compiler object code that is usually neither viewed nor understood by the high-level language programmer.

All of the examples given in this paper were found in real programs. Many were generated by contemporary optimizing compilers.

Although IBM System/370 instructions are used in the examples, these guidelines are believed to be applicable to most register-based architectures.

INTRODUCTION TO PIPELINED PROCESSORS

Most assembly language programmers view each machine instruction as a primitive, indivisible operation. This is an entirely appropriate view for those programmers who mainly use a higher level language and who understand the machine architecture for the purposes of debugging, interfacing with other languages or systems, or manipulating special hardware facilities.

However, for those programmers who use assembly language in order to produce programs of maximum efficiency, an understanding of the implementation of the instruction set may be desirable.

STEPS IN THE PROCESSING OF AN INSTRUCTION

Conceptually, the implementation of even the simplest instruction consists of several sequential steps. For example, consider an Add instruction that fetches an operand from storage and adds it to the contents of an operand register (c[R] <- c[R] + c[M]). To implement such an instruction, a processor would perform the following steps:

1. Compute the address of the instruction. This can be done by adding together the address and length of the previous sequential instruction, or by computing the operand address of a branch-type instruction.

2. Fetch the instruction from storage.

3. Decode the instruction. Recognize that it is an Add instruction, that it has a storage operand, that it alters a register but not storage, etc.

4. Compute the address of the operand of the instruction.

5. Fetch the operand from storage.

6. Execute the instruction. (Add the operand values.)

7. Put away the results of the instruction. (Store the sum back into the register, set the condition code, etc.)

These steps can be depicted on a timing diagram as follows:

```
COMPUTE                    COMPUTE
INSN     FETCH   DECODE    OPND      FETCH   EXECUTE  PUTAWAY
ADDRESS  INSN    INSN      ADDRESS   OPND    INSN     RESULTS
|--------+-------+---------+---------+-------+--------+-------|
   IA       IF      ID        OA        OF      EX       PA
```

The timing diagrams used throughout this paper assume an idealized processor that executes these same steps for each instruction.

ASSEMBLY LINE PARALLELISM

Simple processors execute instructions serially. That is, they complete all the steps involved in processing an instruction before going on to process the next instruction. This minimizes the amount of hardware required to implement an instruction set, but also limits performance.

```
                 IA IF ID OA OF EX PA
Instruction N    |-------------------|
                                      IA IF ID OA OF EX PA
Instruction N+1                       |-------------------|
```

To obtain higher performance, more complex processors employ additional hardware to process multiple instructions concurrently. This parallel processing can be attained in several ways. For example, certain hardware facilities, such as floating-point adders, can be duplicated to permit their simultaneous use by otherwise sequential instructions.

Another approach to parallel instruction processing is to arrange the facilities of the processor in the form of an assembly line. For each of the steps necessary to process an instruction there might exist a specialized facility. To perform an instruction, the processor would pass it from one facility to the next until all of the required steps have been performed. At any given moment there might be several instructions in progress, each occupying a different facility.

```
                  IA IF ID OA OF EX PA
Instruction N     |--+--+--+--+--+--+--|
                     IA IF ID OA OF EX PA
Instruction N+1      |--+--+--+--+--+--+--|
                        IA IF ID OA OF EX PA
Instruction N+2         |--+--+--+--+--+--+--|
                           IA IF ID OA OF EX PA
Instruction N+3            |--+--+--+--+--+--+--|
```

Note that each instruction takes just as long to complete as in a simple, unoverlapped processor. However, the rate at which instructions are completed is increased substantially.

PIPELINED IMPLEMENTATIONS

The preceding discussion of assembly line parallelism assumed that each step in the processing of an instruction required exactly one unit of time. Real pipelined implementations deviate from this model.

For most real processors, time is measured in units known as machine cycles, or cycles for short. The pipeline, then, consists of segments that are each one cycle (or a fixed number of cycles) in duration. The examples in this section illustrate some of the different ways in which the conceptual steps in the processing of an instruction can be mapped onto a pipelined hardware structure.

CODING PRACTICES THAT DEGRADE PIPELINE PERFORMANCE

Each of the coding practices discussed in this section reduces the effectiveness of an instruction pipeline by creating disruptions, or gaps, in the flow of instructions. In each case, the disruption results from a condition in which an instruction must wait in an early stage of the pipeline until some previous instruction leaves a later stage of the pipeline.

ADDRESS GENERATION INTERLOCKS

An address generation interlock (AGI) condition exists whenever an instruction requires a general purpose register for its operand address calculation but the register is unavailable because it will be written by some preceding, unexecuted instruction. Figure 1 illustrates a simple AGI between two consecutive Load instructions. As shown, the operand address calculation step (OA) of the second instruction must be delayed until the putaway-results step (PA) of the first instruction. The delay results in a performance loss that, for typical implementations, can be comparable in magnitude to the execution time of a simple instruction (such as Load).

```
L      R3,INDEX          ID OA OF EX PA
L      R4,VECTOR(R3)        ID       OA OF EX PA
                         1  2  3  4  5  6  7  8
```

Figure 1. AGI Between Consecutive Load Instructions

The obvious way to reduce or eliminate an AGI is to separate the pair of instructions that cause it. While this may not be possible in all cases, it is possible in many common situations. For example, consider the instruction sequence shown in Figure 2. It is is a commonly used implementation of a subroutine call, and is in fact generated by several popular FORTRAN, PL/I and COBOL compilers. Note that an AGI exists between the instruction (L) that loads register R15 with the subroutine address and the next instruction (BALR) which links to the subroutine. If the Load Address instruction (LA), which calculates the address of the subroutine parameter list, were exchanged with the Load instruction, its execution would overlap the AGI condition. In effect, it would execute "for free" during the AGI gap, as shown in Figure 3.

```
LA     R1,ARGLIST        ID OA OF EX PA
L      R15,=A(SUBR)         ID OA OF EX PA
BALR   R14,R15                 ID       OA OF EX PA
                         1  2  3  4  5  6  7  8  9
```

Figure 2. AGI between Load and Branch and Link

```
L      R15,=A(SUBR)      ID OA OF EX PA
LA     R1,ARGLIST           ID OA OF EX PA
BALR   R14,R15                 ID    OA OF EX PA
                         1  2  3  4  5  6  7  8
```

Figure 3. AGI Reduced by Separating Instructions

Another example may be found in a typical implementation of the PL/I INDEX statement. Figure 4 shows compiler generated code that searches a character string for the first occurrence of a specific character pattern. In this case, the pattern is "*/". The code increments a pointer register (R3) on each iteration, and examines the pointed-to characters with a Storage-to-Storage Compare instruction (CLC). Note that the Compare instruction requires two operand address calculation steps (OA), and that the AGI exists between the indexing instruction (BXLE) and the first of the two OA steps for the CLC. Since the compare operation is commutative, the operands of the CLC could be reversed, thereby overlapping the AGI with the processing of the first CLC operand, as shown in Figure 5.

```
LOOP  CLC    0(2,R3),=C'*/'   COMPARE FIELD WITH LITERAL
      BE     FOUNDIT          BRANCH ON EQUALITY
      BXLE   R3,R5,LOOP       INCREMENT R3 AND ITERATE

Timing:  BXLE                 ID OA OF EX PA
         CLC Operand 1           ID       OA OF EX PA
         CLC Operand 2                    ID OA OF EX PA
         BE                                  ID OA OF EX PA
                              1  2  3  4  5  6  7  8  9  10
```

Figure 4. AGI with First Operand of Compare Instruction

```
LOOP  CLC    =C'*/',0(R3)     COMPARE LITERAL WITH FIELD
      BE     FOUNDIT          BRANCH ON EQUALITY
      BXLE   R3,R5,LOOP       INCREMENT R3 AND ITERATE

Timing:  BXLE                 ID OA OF EX PA
         CLC Operand 1           ID OA OF EX PA
         CLC Operand 2              ID    OA OF EX PA
         BE                                ID OA OF EX PA
                              1  2  3  4  5  6  7  8  9
```

Figure 5. AGI With Second Operand of Compare Instruction

In Figure 5, note that the pipeline gap is only partly filled by the processing of the first operand of the CLC instruction. For the gap to be completely filled, the processor would have to have implemented the ID, OA and OF steps in only two machine cycles, which is usually impractical.

Therefore, the utilization of the pipeline could be further increased by interposing another instruction between the BXLE and the CLC. Figure 6 shows a recoding of the loop which makes use of the BE instruction to further fill the pipeline gap. However, while it improves pipeline performance, it does so at the expense of the space and time of two additional instructions. This trade-off could be worthwhile on a pipelined processor provided that the code generally looped at least once. On a nonpipelined processor, the code of Figure 5 would be preferable.

```
      B      COMP             BRANCH INTO LOOP
LOOP  BE     FOUNDIT          BRANCH OUT ON EQUALITY
COMP  CLC    =C'*/',0(R3)     COMPARE LITERAL WITH FIELD
      BXLE   R3,R5,LOOP       INCREMENT R3 AND ITERATE
      BE     FOUNDIT          HANDLE FALL THRU CASE

Timing:  BXLE                 ID OA OF EX PA
         BE                      ID OA OF EX PA
         CLC Operand 1              ID OA OF EX PA
         CLC Operand 2                 ID OA OF EX PA
                              1  2  3  4  5  6  7  8
```

Figure 6. AGI Eliminated by Reordering

Another example of a reducible AGI penalty is shown in Figure 7. A simple AGI exists between the instruction that loads a pointer and the next instruction which uses the pointer. As shown in Figure 8, the AGI penalty can be reduced by changing the code so that it fetches the constant before fetching the indexed data operand.

```
L      R3,POINTER        ID OA OF EX PA
L      R4,DATA(R3)          ID       OA OF EX PA
A      R4,CONSTANT                   ID OA OF EX PA
ST     R4,DATA(R3)                      ID OA OF EX PA
                         1  2  3  4  5  6  7  8  9  10
```

Figure 7. AGI Due to Use of Pointer

```
L       R3,POINTER          ID OA OF EX PA
L       R4,CONSTANT            ID OA OF EX PA
A       R4,DATA(R3)               ID    OA OF EX PA
ST      R4,DATA(R3)                        ID OA OF EX PA
                            1  2  3  4  5  6  7  8  9
```

Figure 8. AGI Reduced by Separating Load

OPERAND FETCH INTERLOCKS

An operand fetch interlock (OFI) condition exists whenever an instruction requires an operand from storage but the operand is unavailable because it will be modified by some preceding, unexecuted instruction. Consider the code shown in Figure 9, which was found in a major control program. The move instruction (MVC) stores into its operand (WORKAREA) during the last step of its execution, and the subsequent Compare instruction (CLI) needs to fetch from the modified storage area. Consequently, the CLI instruction is unable to perform its operand fetch step (OF) until after the the MVC instruction completes its putaway step (PA), resulting in a substantial pipeline disruption.

```
MVC      WORKAREA(8),USERVAR   MOVE USER DATA TO TEMP
CLI      WORKAREA+7,C' '       COMPARE LITERAL TO TEMP

Timing:  MVC Operand 1         ID OA OF
         MVC Operand 2            ID OA OF EX PA
         CLI                         ID OA       OF EX PA
                               1  2  3  4  5  6  7  8  9
```

Figure 9. Operand Fetch Interlock (OFI)

By modifying the Compare instruction so that it fetches its needed data from the source operand of the Move instruction, rather than from the destination operand, the OFI can be eliminated entirely. This is illustrated in Figure 10.

```
MVC      WORKAREA(8),USERVAR   MOVE USER DATA TO TEMP
CLI      USERVAR+7,C' '        COMPARE LITERAL TO USER DATA

Timing:  MVC Operand 1         ID OA OF
         MVC Operand 2            ID OA OF EX PA
         CLI                         ID OA OF EX PA
                               1  2  3  4  5  6  7
```

Figure 10. OFI Eliminated by Refetching User Data

Figure 11 shows a commonly occuring OFI that can often be reduced, if not eliminated. The And Immediate instruction (NI) performs an implicit fetch of its operand (FLAGS) before storing its result. The fetch is delayed until the store is performed by the preceeding Or Immediate instruction (OI). As shown in Figure 12, the unrelated Move instruction (MVC) can be interposed and thereby overlapped with the OFI delay.

```
MVC      VARA(8),VARB     UNRELATED MOVE INSTRUCTION
OI       FLAGS,X'80'      SET BIT IN BYTE OF FLAGS
NI       FLAGS,X'FB'      CLEAR BIT IN BYTE OF FLAGS

Timing:  MVC Operand 1    ID OA OF
         MVC Operand 2       ID OA OF EX PA
         OI                     ID OA OF EX PA
         NI                        ID OA       OF EX PA
                          1  2  3  4  5  6  7  8  9  10
```

Figure 11. OFI Due to Logical Storage Operations

```
OI       FLAGS,X'80'      SET BIT IN BYTE OF FLAGS
MVC      VARA(8),VARB     UNRELATED MOVE INSTRUCTION
NI       FLAGS,X'FB'      CLEAR BIT IN BYTE OF FLAGS

Timing:  OI                    ID OA OF EX PA
         MVC Operand 1            ID OA OF
         MVC Operand 2               ID OA OF EX PA
         NI                             ID OA OF EX PA
                               1  2  3  4  5  6  7  8
```

Figure 12. OFI Eliminated by Separating Instructions

Another common sequence involves the construction of parameter lists for calls to subroutines. A standard convention is to pass a table of full words that contain the addresses of the individual parameters. The end of the parameter list is indicated by setting bit 0 of the last full word in the table, as shown in Figure 13.

```
LA      R3,ARGUMENT       ID OA OF EX PA
ST      R3,ARGLIST           ID OA OF EX PA
OI      ARGLIST,X'80'           ID OA       OF EX PA
                          1  2  3  4  5  6  7  8  9
```

Figure 13. OFI Due to Logical Storage Operation

As with the previous examples, an OFI exists between the store instruction that stores the last address into the table and the Or Immediate instruction (OI) that sets the end-of-table flag. This OFI could be eliminated in either of two ways. One is to replace the Or Immediate instruction with a Store-Type instruction (such as MVI) that does not require a fetch of its operand. See Figure 14. The other is to perform the setting of the flag in the processor registers rather than in storage, as depicted in Figure 15.

```
LA      R3,ARGUMENT       ID OA OF EX PA
ST      R3,ARGLIST           ID OA OF EX PA
MVI     ARGLIST,X'80'           ID OA OF EX PA
                          1  2  3  4  5  6  7
```

Figure 14. OFI Eliminated by Using a Move Instruction Instead of an Or Instruction

```
LA      R3,ARGUMENT       ID OA OF EX PA
O       R3,=X'80000000'      ID OA OF EX PA
ST      R3,ARGLIST              ID OA OF EX PA
                          1  2  3  4  5  6  7
```

Figure 15. OFI Eliminated by Performing the Or Operation in a Register

The code in Figure 15 requires four additional bytes of storage to hold the literal value, but a single instance of the literal could be shared by several copies of the executable code.

INSTRUCTION FETCH INTERLOCKS

The last step in the execution of a Store-Type instruction is the alteration of the operand field in storage. If a Store-Type instruction alters one of its immediate successors, then the processor cannot obtain that successor instruction and start it through the pipeline until it has been altered. The resulting degradation is illustrated in Figure 16.

```
         LH     R3,SCON(INDEX)     GET S-TYPE ADCON FROM TABLE
         STH    R3,BDFIELD         MODIFY BRANCH TARGET ADDRESS
         B      *-*                N-WAY BRANCH
BDFIELD  EQU    *-2                BASE,DISPLACEMENT

   Timing:  LH        ID OA OF EX PA
            STH          ID OA OF EX PA
            B                        IA IF ID OA OF EX PA
                       1  2  3  4  5  6  7  8  9  10 11 12
```

Figure 16. Instruction Fetch Interlock Due to Self-Modifying Code

Fortunately, self-modifying programs have been recognized as being undesirable for other reasons and are relatively rare. But even if the instruction code is not self-modifying, it may appear to be so if it is intermixed with data variables.

Figure 17 shows an instruction sequence that builds an inline parameter list as part of a subroutine call. Although the code will branch around the modified parameter list, a pipelined processor that prefetches instructions will have great difficulty in recognizing this fact. The performance impact depends upon how and when the processor detects this apparent but falsely detected store into the instruction stream. Typically, the processor will need to discard and then refetch some number of prefetched instructions when the store instructions are completed.

```
          L      R3,ARG1
          ST     R3,ARGLIST
          L      R3,ARG2
          ST     R3,ARGLIST+4
          CNOP   0,4
          BAL    R1,SUBRTN
RETADDR   DC     A(NEXTINSN)
ARGLIST   DS     2F
NEXTINSN  EQU    *
```

Figure 17. Instruction Fetch Interlock Due to Inline Parameter List

UNNECESSARY TAKEN BRANCHES

Branch instructions are "bad news" to a pipelined processor. They tend to disrupt the natural flow of instructions through the pipeline by reducing the processors ability to prefetch the instructions that it needs.

Processors may use sophisticated algorithms to anticipate which instructions will be executed, but even the most complex of these techniques cannot eliminate the cost of branching but can only reduce it.

Although all branches are troublesome, taken branches are the most costly. Wherever possible, code should be arranged so that the most frequently executed paths tend to fall through conditional branch instructions. Figure 18 shows an example of some code that normally branches around a call to an error routine. To make the best use of a pipelined processor, it would be better if the call to the error routine were located elsewhere in the program and the mainline code branched to the call when there was an error, as depicted in Figure 19.

```
      TM     FLAG,X'04'    TEST FLAG IN STORAGE
      BZ     OK            BRANCH IF SITUATION NORMAL
      CALL   ERROR         INVOKE ERROR HANDLING PROCEDURE
      B      EXIT
OK    DS     0H
```

Figure 18. Branching Around Inline Exception-Handling Code

```
       TM     FLAG,X'04'    TEST FLAG IN STORAGE
       BNZ    NOTOK         BRANCH IF EXCEPTIONAL CASE
OK     DS     0H

NOTOK  CALL   ERROR         INVOKE ERROR HANDLING PROCEDURE
       B      EXIT
```

Figure 19. Branch to Out-of-Line Exception-Handling Code

There is one case in which it is slightly better to have the most frequently used path be the taken leg of a conditional branch. As shown in Figure 20, this is the case in which the code forks in two directions and there is no real fall-through path. Whether or not the conditional branch instruction is taken, the processor will experience a discontinuity in its fetching of instructions. On most processors known to the author, it is better if LABELX is on the more frequently used path. On some processors, it makes no difference. In no case is it preferable for LABELY to be on the more frequently used path.

```
TM     FLAG,X'80'
BZ     LABELX        SHOULD BE THE MORE USED PATH
B      LABELY        SHOULD BE THE LESS USED PATH
```

Figure 20. Situation where Neither Path Falls through

EXECUTION INTERLOCKS

For very simple instruction sets, it is easy to organize the execution facilities of the processor in the form of a hardware pipeline. Then the execution facilities can accept a new instruction (from the instruction preprocessing facilities) on every cycle.

For most real instruction sets, such as that of the System/370 architecture, there are complex instructions whose execution requires a substantial process of iteration. (Examples might include floating-point multiplication and division instructions such as MD and DD, or data translation instructions such as TR and TRT.) Such instructions cannot be easily pipelined since pipelining amounts to the unraveling of the iterations onto a hardware assembly line. Instead, most implementations allow the execution facilities to become busy and hold up the pipeline while the execution proceeds iteratively.

To minimize the performance loss due to execution facility hold ups, the programmer requires a knowledge of the internal capabilities of the processors on which his or her programs will run. But even without specific model dependent information, one can make reasonable programming assumptions.

For instance, consider the loop shown in Figure 21 which is widely used to perform the decomposition of a matrix by implementing the iteration: C(I) = Z*A(I) + B(I), where A, B and C are vectors and Z is a constant [1]. Note that there are two sources of possible pipeline degradation in the loop: (1) the AGI that exists between the BXLE instruction and the LD instruction, and (2) the AD and MDR instructions which will create an execution bottleneck for most processor implementations. The timing diagram for Figure 21 assumes that the MDR and AD instructions each require eight and four execution busy cycles, respectively.

```
LOOP  LD    R4,A(RX)     FETCH A(I)
      MDR   R4,R2        MULTIPLY BY CONSTANT Z
      AD    R4,B(RX)     ADD B(I)
      STD   R4,C(RX)     STORE RESULT INTO C(I)
      BXLE  RX,R6,LOOP   INCREMENT RX AND ITERATE

Timing:

 BXLE     ID OA OF EX PA
 LD          ID       OA OF EX PA
 MDR            ID       OA OF E1 E2 E3 ... E8 PA
 AD                ID       OA OF             E1 E2 E3 E4 PA
 STD                  ID       OA OF                      EX PA
          1  2  3  4  5  6  7  8  9  10 ... 15 16 17 18 19 20 21
```

Figure 21. Simple Symmetric Decomposition Loop

By recoding the first two instructions of the loop as shown in Figure 22, it is possible to partially reduce the AGI penalty.

```
LOOP  LDR   R4,R2        COPY CONSTANT Z
      MD    R4,A(RX)     MULTIPLY BY A(I)
      AD    R4,B(RX)     ADD B(I)
      STD   R4,C(RX)     STORE RESULT INTO C(I)
      BXLE  RX,R6,LOOP   INCREMENT RX AND ITERATE

Timing:

 BXLE     ID OA OF EX PA
 LDR         ID OA OF EX PA
 MDR            ID    OA OF E1 E2 E3 ... E8 PA
 AD                ID    OA OF             E1 E2 E3 E4 PA
 STD                  ID    OA OF                      EX PA
          1  2  3  4  5  6  7  8  9  10 ... 15 16 17 18 19 20
```

Figure 22. Improved Loop with Reduced AGI

By further reordering the loop as shown in Figure 23, one can totally overlap the AGI delay with the execution delay for the MDR instruction. The cost of this improvement is one extra Load instruction outside of the loop. There is also a potential problem in that the last iteration of this loop will make an unwanted storage access beyond the last element of vector A, and could cause an extraneous access exception.

```
      LD    R4,A(0)      FETCH A(0)
LOOP  MDR   R4,R2        MULTIPLY BY CONSTANT Z
      AD    R4,B(RX)     ADD B(I)
      STD   R4,C(RX)     STORE RESULT INTO C(I)
      LD    R4,A+8(RX)   FETCH A(I+1)
      BXLE  RX,R6,LOOP   INCREMENT RX AND ITERATE

Timing:

 BXLE     ID OA OF EX PA
 MDR         ID OA OF E1 E2 E3 ... E8 PA
 AD             ID    OA OF           E1 E2 E3 E4 PA
 STD               ID    OA OF                    EX PA
 LD                   ID    OA OF                    EX PA
          1  2  3  4  5  6  7  ... 12 13 14 15 16 17 18 19
```

Figure 23. Improved Loop with Eliminated AGI

LATE CONDITION CODE SETTING

On some pipelined machines (for example, the S/370 Model 195), the processor can make use of early knowledge of the condition code to reduce the amount of work that the processor must perform. If the condition code is already known at the time that a conditional branch instruction is decoded, the processor may be able to resolve the branch at decode time (during its DA step) rather than at execution time. This allows the processor to treat the branch as either unconditionally taken or not taken. There is thus no need to prefetch instructions from both paths, and there is no chance that a wrong branch guess will require that preprocessed instructions be discarded.

To permit a pipelined processor to resolve conditional branch instructions as early as possible, the programmer should attempt to separate the condition code setting instruction from the conditional branch. This is analogous to the treatment for AGIs and, as with AGIs, cannot be done in many cases.

OTHER CODING PRACTICES THAT DEGRADE PERFORMANCE

Unlike the coding guidelines for pipelined processors, the guidelines given in this section apply equally well to many nonpipelined processors.

DISCONNECTED ARRAYS

Every modern pipelined processor known to the author employs a pipelined cache memory to help match the cycle time of the storage system to that of the processor logic. Most programmers are familiar with the software technique of paging and can understand the operation of a cache by direct analogy: the cache is to main memory as main memory is to the backing-store of a paged memory system.

With software paging, the best performance is attained when the programmer minimizes the number of pages that the program requires coresident in main storage, or, more precisely, when the programmer minimizes the working set size of the program. The same principle applies to caches. The programmer should try to maximize the locality of his or her programs to avoid unnecessary traffic between the cache and main storage.

For the average program, there is relatively little that the programmer might do to significantly affect the performance of the cache. The hardware algorithms for cache management are usually very effective. However, for programs that operate upon large volumes of data per unit of computation, the cache can become overloaded and experience a performance degradation analogous to "thrashing."

Numerical programs that manipulate large arrays of data are particularly susceptible to performance problems. If such a program accesses a large disconnected array, or cross section of an array, it may cause blocks of data to be loaded into the cache at a very high rate. The rate may get so high that a newly loaded block often gets replaced by another block prior to be being reused. To reduce the likelihood of this thrashing phenomenon, array processing programs should be coded to make sequential accesses to storage wherever possible.

UNALIGNED BRANCH TARGETS

When a branch instruction is taken, the processor must begin to process instructions starting at the location specified by the branch target address. To obtain the required instructions, the processor will initiate a storage fetch operation which will return some number of bytes of instructions. On most contemporary machines, the fetch will return the doubleword that contains the branch target location.

If the branch target location happens to be the first byte of a doubleword, the initial branch target fetch can yield eight bytes of usable instructions. But if the target location is the last halfword of a doubleword, then only two usable bytes are obtained and the processor is more likely to run out of instructions to decode.

Thus, to improve performance on pipelined (and even on many nonpipelined) processors, it is advantageous to align the targets of frequently executed branch instructions on doubleword boundaries. This is especially worthwhile for high frequency loops since the extra storage space expended to accomplish the alignment is small relative to the frequency of use.

UNALIGNED DATA

Most modern processors feature special hardware that efficiently processes unaligned data. Thus, if the four-byte operand of a Load instruction is not placed on a fullword boundary, the machine will automatically preshift the data at no increase in execution time. However, if the unaligned operand happens to cross a doubleword boundary (assuming that a doubleword is the fundamental unit of data transfer in the processor), there is an inherent performance degradation because of the need to fetch two doublewords rather than one. The increase in execution time for such a Load instruction can easily be on the order of a factor of two.

WHY SOME "SAVED" CYCLES ARE NOT REALLY SAVED

The guidelines stated in this paper assume that by removing program interlocks one can increase the utilization of a pipelined processor and thereby improve performance. While this is generally true, it does not follow that an improvement will be attained in each instance.

Statistics indicate that, in most situations, much of the expected performance improvement will be achieved. Thus, it should be understood that the static guidelines given in this paper are not certain, but only likely, to produce the desired improvement in the complex, dynamic flow of instructions through the processor.

SIMULTANEOUS INTERLOCKS

In any given cycle, there may be many instructions active in a pipelined processor. As a result, there may be more than one reason for a gap in the flow of instructions through the pipeline. Removing one of the interlocks may not result in higher pipeline utilization.

PREFETCHING MAKES USE OF IDLE CYCLES

Not all gaps in the pipeline are wasted. Depending upon the design of the processor, pipeline gaps may be used for anticipatory prefetching of instructions or data into fast buffer registers. It can happen that the elimination of a pipeline gap will later result in a delay because of empty instruction or data buffers.

E-UNIT CONGESTION

For simple instructions, the E-unit hardware can easily be pipelined as an extension of the I-unit pipeline. For more complex instructions, such as Divide or Edit And Mark, the E-unit may require a number of iterative execution cycles, and thus may not be able to keep up with the I-unit pipeline. When this happens, the E-unit is the bottleneck, and there may be little to be gained by increasing the utilization of the pipeline.

CONCLUSION

The efficiency of a pipelined processor is highly dependent upon interactions among the instructions that it executes. While there are many unique pipeline implementations, the intrinsic steps in the processing of an instruction lead to a set of guidelines that are broadly applicable. Code generation, whether manual or automated, should be sensitive to these performance considerations.

REFERENCES

1. Michael N. Gemmel, "Techniques for Improved Performance of Loops in Very Large IBM Processors," Technical Report 00.3038, IBM Corporation, Poughkeepsie, New York, February 26, 1980.

BIBLIOGRAPHY

1. IBM System/360 Model 195, Functional Characteristics, Form A22-6943-0, IBM Corporation, Poughkeepsie, New York, August 1969.

2. IBM System/360 Model 195, Theory of Operation: System Introduction and Instruction Processor, Form SY22-6855-0, IBM Corporation, Poughkeepsie, New York, August 1970.

3. IBM System/370 Model 168, Theory of Operation/Diagrams Manual, Volume 2, I-unit, Form SY22-6932-0, IBM Corporation, Poughkeepsie, New York, April 1973.

4. IBM 3033 Processor Complex, Theory of Operation/Diagrams Manual, Volume 2, Instruction Preprocessing Function (IPPF), Form SY22-7002-0, IBM Corporation, Poughkeepsie, New York, January 1978.

5. Ramamoorthy, C. V., and Li, H. F., "Pipeline Architecture," ACM Computing Surveys, Volume 9, Number 1, March 1977, pp. 61-102.

6. Martin, Daniel B., "Coding Hints for Large Systems," Technical Report 00.1968, IBM Corporation, Poughkeepsie, New York, December 15, 1969.

7. Morris, Derrick, and Ibbett, Roland N., "The MU5 Computer System," Springer-Verlag, New York, 1979.