Proc. 1984 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, August 21-24, 1984, Cambridge, Mass.
Also, Performance Evaluation Review, 12, 3, August, 1984.

Program Optimization for a Pipelined Machine
A Case Study

John Sanguinetti
Amdahl Corporation

Abstract: The Amdahl 580 processor is a pipelined processor whose performance can be affected by characteristics of the instructions it executes. This paper describes certain optimizations made to a set of system software routines during their development. The optimization effort was driven by the execution frequencies of common paths through the programs in question, and by the execution characteristics of those paths, as shown by a processor simulator. Path optimization itself was done with both general program optimization techniques and with techniques specific to the particular characteristics of the 580's pipeline. Overall, the average execution time for these routines was reduced by over 50%.

1.0 Introduction

The performance of a pipelined central processor typically varies depending on the instruction sequence of the program being executed. A significant performance improvement can often be made to a given sequence of instructions by altering that sequence to take advantage of pipeline characteristics. Overall improvement to the performance of a program can be effected by optimizing the most frequently occurring instruction sequences, or paths. This paper is a report on a project which optimized a particular set of system software routines which execute on the Amdahl 580 series of computers. The optimizations were made taking into account the pipeline characteristics of the 580 central processor and they resulted in significant performance improvement. Because this project was done by the manufacturer of the target processor, detailed information about the characteristics of the processor were available.

The software in question is a small piece of a larger system which was designed and implemented in the same manner as many other modern software development projects, with the designers producing a functionally correct implementation first, and optimizing its performance second. The techniques and tools used to improve the performance of the initial implementation are generally applicable to program optimization problems, though some of the optimizations made were specific to the design of the pipeline in the 580 processor.

In order to optimize the set of routines, path frequencies were required to know which paths were the most important. This required meaningful workloads which could be instrumented to determine the frequency with which each routine was executed, and, within each routine, which paths were executed. Of course, different workloads generate different path frequencies, so the result of this step is a set of frequencies for each workload with which to evaluate the performance.

Once the path frequencies were identified, their execution times were needed. Ultimately, path times and frequencies are all that is necessary to determine the total amount of time the routines consume. However, in order to make improvements in the paths, more information about their execution characteristics is needed. In order to provide this information, a program which simulates the behavior of the processor was used. The input to this simulator is an instruction sequence, complete with operand addresses and values, and the output is a detailed description of the flow of this sequence through the processor pipeline. Identification of delays due to storage contention and pipeline interlocks are especially useful in optimizing an instruction sequence.

Finally, in order to obtain an overall figure of merit to be used to evaluate each successive refinement of this software, a weighted average of time per path was obtained. This was simply the time of each path weighted by the path's frequency. Thus, a different weighted average was obtained for each workload. Because the processor simulator describes execution characteristics in terms of cycles, time is expressed here as number of cycles (the cycle time in the Amdahl 580 processor is 23.25 ns).

The method for evaluating the subject software consisted, then, of the following steps:

1) obtain path frequencies for the routines for several different workloads.

2) generate an instruction sequence for each path whose frequency is significant.

3) use the simulator to obtain execution characteristics of each path.

4) use the path times and frequencies to produce a weighted average of time/path to use as an overall figure of merit for each workload.

Once the performance of the software in question was evaluated, it could be optimized in a relatively straightforward manner. This process consisted of selecting a path to be optimized and then analyzing its execution characteristics. A path's execution time was improved by applying the techniques described in section 4.

2.0 Obtaining path frequencies

Path frequency information is vital to a performance evaluation of a piece of software, and likewise, it is vital to an optimization effort. Typically, a small number of paths accounts for the majority of the execution time (the 90-10 rule is nearly universal, and this case was no exception), and these are the paths which most of the effort should be concentrated on.

Path frequencies, of course, are workload dependent, and thus must be derived from workloads which have some relationship to the ultimate production environment. Because this is system software, it will be used by a wide variety of workloads. We chose three workloads that represent batch, time-sharing, and transaction processing workloads.

```
     address instruction        int.   pipeline flow
 1   048066  LR   1806 - - - - - - -  GBLEW
 2   048068  N    5400B220 - - - - - GBLEW
 3   04806C  BC   4770C14A - - CCI -  GGBLEW
 4   048070  L    5820800C - - - - - -  GBLEW
 5   048074  LA   41300020 - - IFCH- - -  GBLEW
 6   048078  ALR  1E32 - - - - EEI - - - - GBBLEW
 7   04807A  LR   1821 - - - - - - - - - - -  GGBLEW
 8   04807C  N    5420B2BC - - - - - - - - - -  GBLEW
 9   048080  LA   41F00020 - - - - - - - - - - GBLEW
10   048084  SRL  88F00002 - - - - - - - - - -  GBLEEEEW
11   048088  ALR  1E4F - - - - EEI - - - - - - - GBBBBBLEW
12   04808A  ALR  1E48 - - - - EEI - - - - - - -  GGGGGBBLEW
13   04808C  MVC  D2034000D040 EGI - - - - - - - - - - - GGGGGBLEW
14   04808C  MVC  D2034000D040 - - - - - - - - - - - - - -  GBLEW
15   04808C  MVC  D2034000D040    - - - - - - - - - -  - - - GBLEW
```

explanation:

* The 580 central processor has a 5 stage pipeline:
  - G - address generation
  - B - high-speed buffer access
  - L - logical testing
  - E - execution unit cycle
  - W - write results

* Each line in the picture represents a pipeline flow. Each character position in the pipeline flow represents one cycle. Delays are represented by repetition of the cycle letter in which the delay occurs.

* Some instructions (e.g. 10) require several cycles in the E-stage.

* Some instructions (e.g. 13) are implemented by more than one flow through the pipe.

* The type of delay is labelled:
  - CCI - condition-code interlock
  - EEI - execute-execute interlock
  - EGI - execute-generate interlock
  - IFCH - instruction fetch delay

  If more than one type of delay affects an instruction, the longer one is indicated.

* This sequence takes 30 cycles.

Picture of the Pipeline Flow of an Instruction Sequence
Figure 1

Actually obtaining frequencies for these routines is simply a matter of running the workload in question and counting each occurrence. Of course, there are a number of complications. Since the code itself could be easily instrumented, obtaining the path frequencies was relatively easy after the system under development progressed to the point that the workloads could be run. Before the system was operational, path frequencies could only be obtained by extrapolating from data obtained on other machines using different operating software.

3.0 Obtaining path lengths

In order to obtain the length, and other execution characteristics, of a path, an instruction sequence for the path had to be generated. This was done by running the routines in a virtual machine (a modification of VM/370) that had the capability to trace instructions. A test program was used which could force the various routines through the desired paths. When the test program was executed with tracing active, the desired instruction sequence for each path was obtained.

We should note here that, though the instruction sequence for any given path is unique, the operands for each instruction in the path are not, and their variation can affect the execution characteristics of the path. The routines being optimized here have relatively few operands whose use is quite regular, however, so this is a second-order effect, and consequently was ignored.

Once the instruction sequence was obtained, it could be used directly by the processor simulator to give its execution characteristics. Because pipeline processors have quite complicated execution characteristics, a simulator is necessary to understand the execution behavior of a given instruction sequence — see [7] for a discussion of pipeline processors. The simulator produces the overall number of cycles required by the instruction sequence, along with detailed breakdowns of cycles by instruction, delay type, etc. For code optimization purposes, the most useful output is a 2-dimensional picture of the pipeline activity, as shown in figure 1. This picture makes particularly inefficient sequences of code obvious, highlighting where the code needs to be improved. As an example, figure 2 has an instruction sequence which is functionally identical to the sequence in figure 1, but the instructions have been rearranged to improve its pipeline execution behavior. Note that all of the interlocks have been avoided — a fairly unusual case.

4.0 Optimizing techniques

Selecting the paths to be optimized in the proper order is easy, given the frequency and path length information. Doing the optimization itself, of course, is not so easy. It was found that applying a few basic strategies was effective in optimizing most of the paths. We assume here that there are no glaring inefficiencies in the algorithms used. In fact, all of these routines were quite straightforward.

```
      address instruction          int.   pipeline flow
 1    048066  LA   41F00020 - - - - -  GBLEW
 2    04806A  LR   1806 - - - - - - - - GBLEW
 3    04806C  LA   41300020 - - - - - -  GBLEW
 4    048070  ALR  1E48 - - - - - - - - - GBLEW
 5    048072  SRL  88F00002 - - - - - - -  GBLEEEEW
 6    048076  N    5400B220 - - - - - - - - GBBBBLEW
 7    04807A  L    5820800C - - - - - - - -  GGGGBLEW
 8    04807E  BC   4770C14A - - - - - - - - - -  GBLEW
 9    048082  ALR  1E4F - - - - - - - - - - - - - GBLEW
10    048084  L    5800D040 - - - - - - - - - - -  GBLEW
11    048088  ALR  1E32 - - - - - - - - - - - - - - GBLEW
12    04808A  LR   1821 - - - - - - - - - - - - - -  GBLEW
13    04808C  N    5420B2BC - - - - - - - - - - - - - GBLEW
14    048090  ST   50004000 - - - - - - - - - - - - -  GBLEW
```

explanation:

* There are no interlocks in this sequence.

* Most of the instructions have been moved within the sequence.

* The MVC instruction at 13 was replaced by a L and ST at 10 and 14.

* This sequence takes 21 cycles.

A Modification of the Sequence in Figure 1
Figure 2

The routines which were optimized in this project are all fairly small, but have reasonably high frequencies. At the start of the optimization effort, the 17 most common paths ranged in execution time from 109 cycles to 994. In the batch workload, the weighted average was 278.6. (Note that we are concerned here with processor cycles only. Buffer misses, which add considerably to the cycles required, are considered separately.) Thus, small changes in path lengths were generally significant — each cycle saved would usually represent 1/2% to 1% of the path — and a number of small changes would easily add up to significant improvements. By the end of the optimization effort, the range had been reduced to 50 to 360, with the weighted average reduced to 121.8.

4.1 Machine-independent code optimizations

There are a variety of optimization techniques which are more or less machine independent which were used here (see [1]). For other machine-independent optimizations unrelated to the techniques used here, see [2] and [3]. The specific techniques were:

* Optimize common cases

In optimizing a routine, knowledge of the most commonly occurring cases allows tailoring the code to those cases, usually at some cost to the less frequent cases. For example, by assuming that an operand will be a particular value, the time to compute a transformation on that operand can be saved by simply checking the expected value and supplying the precomputed transformation result. Figure 3 contains an example of this type of optimization.

```
original:
    L    R1,PTR       low order 2 bits is the
    LA   R2,3            length
    NR   R2,R1        isolate the coded length
    SLL  R3,0(R2)     multiply by 2**code
    N    R1,=X'FFFFFFFC' remove length
 - - remainder of path - -

optimized:
    TM   PTR+3,3      test for the common case
    BNZ  UNUSUAL      non-zero is unusual case
    L    R1,PTR       low order 2 bits are 0
 - - remainder of path - -
```

explanation:

* R3 contains a displacement to be multiplied by 1, 2, 4, or 8.

* PTR contains an address and an encoded length.

* The original code is replicated at label UNUSUAL.

* The normal case has been reduced by at least 5 cycles, while the unusual case has been lengthed by 3.

Optimizing a Common Case
Figure 3

* Optimize branches

In any given routine, there are usually several possible paths. Usually, one or two are the most common, while the rest are virtually never executed. In a pipelined processor, there is usually a penalty associated with branches which are taken (as opposed to conditional branches which fall through). This is due to the necessity of generating the branch address before being able to begin instruction fetch, and usually amounts to one or two cycles (in the 580, it is one cycle). Consequently, it is advantageous to make the most common path "fall through" — i.e. contain no taken branches. See figure 4 for an example. (See [4] for a more sophisticated method for optimizing branches.)

* Optimize subroutine use

Making extensive use of subroutines is generally accepted as a good design practice. However, subroutine linkage nearly always imposes a non-trivial execution cost. Depending on the path length, the subroutine linkage code can occupy a significant number of the required cycles. In the software we started with, a particularly flexible subroutine calling convention had been established and was rigidly adhered to. Even though the use of subroutines was not excessive when looked at from a functional viewpoint, the cost of the subroutine linkage was excessive. By removing all subroutine calls from the common paths, and putting the necessary code inline, path lengths were reduced by an average of 20%.

* Optimize instruction use

In processors like the 580, instruction time differs based on the characteristics of the instruction. Particularly, those instructions which have two operands in storage are more expensive than those with only one in storage and the other in a register. Also, instructions which store results into memory generally cause an extra cycle of delay since writing the result will often collide with (and delay) a subsequent buffer read. Consequently, modifying a routine to avoid these instructions can yield performance improvement. For example, see the replacement of an MVC instruction by a Load and a STore in figures 1 and 2.

4.2 Optimize instruction sequences

This optimization is generally pipeline-dependent and is far more difficult than the other techniques described here. The point here is to improve the instruction flow through the processor's pipeline. This is done primarily by rearranging instructions in the sequence so that interlocks and delays are avoided. In order to do this, an aid like the processor simulator is necessary, since the behavior of the pipeline is quite complicated.

```
original:
        C    R1,OP1
        BNE  CASE2
        L    R2,XYZ          this is the more
        A    R2,ONE            common case
        B    COMMON
CASE2   L    R2,ABC          this is the less
        S    R2,ONE            common case
COMMON - - remainder of path - -

optimized:
        C    R1,OP1
        BNE  CASE2
        L    R2,XYZ          this is the more
        A    R2,ONE            common case
COMMON - - remainder of path - -

CASE2   L    R2,ABC          this is the less
        S    R2,ONE            common case
        B    COMMON
```

explanation:

* The optimized code has no taken branches in the more frequent path. This saves at least 2 cycles.

* The most frequent path is now shorter, and more likely to fit in the high-speed buffer.

Optimizing Taken Branches
Figure 4

There are several classes of pipeline interlocks which are common to most pipeline processors — see [6], [7], and [8] for discussions of pipeline processors and their interlocks. Many interlocks can be avoided, depending on the logic of the program, by rearranging the instruction sequence. See [5] for an algorithm to rearrange code for a simple pipeline.

* CCI - condition code interlock

A CCI interlock occurs when one instruction sets the condition code and the next instruction is a conditional branch. This is a common sequence, whose execution looks like:

```
C     R1,OP          G B L E W
BE    TARGET           G G B L E W
                         |
                     CCI interlock
```

This interlock can be eliminated by inserting another instruction, which does not change the condition code, between the two instructions. In effect, the otherwise wasted cycle is used to do something that would have to be done later:

```
C     R1,OP          G B L E W
L     R2,LOC           G B L E W
BE    TARGET             G B L E W
```

* EEI - execute-execute interlock

An EEI interlock occurs when an operand for the execute stage (E-stage) of the instruction is still being modified by a previous instruction. An example is:

```
A     R1,OPA         G B L E W
S     R1,OPB           G B B L E W
                           |
                      EEI interlock
```

This interlock is eliminated by separating the two instructions with others which do not use the operand in contention. Note that, in this case, it requires 2 intervening instructions (cycles) rather than one:

```
A     R1,OPA         G B L E W
L     R2,LOC1          G B L E W
L     R3,LOC2            G B L E W
S     R1,OPB               G B L E W
```

Again, the interlock is eliminated by making use of the otherwise wasted cycles.

* EGI - execute-generate interlock

An EGI interlock occurs when one instruction computes a value and the next instruction uses that value to generate an address. An example is:

```
A     R1,OP          G B L E W
ST    R2,0(,R1)        G G G G B L E W
                         | | |
                      EGI interlock
```

The solution to this case is the same as for the previous two cases, separate the two contending instructions:

```
A     R1,OP          G B L E W
L     R3,LOC1          G B L E W
L     R4,LOC2            G B L E W
L     R5,LOC3              G B L E W
ST    R2,0(,R1)              G B L E W
```

* SFI - store-fetch interlock

An SFI interlock occurs when one instruction stores a value and the next instruction uses it for an operand. An example is:

```
ST    R2,ABC         G B L E W
A     R1,ABC           G B L L L E W
                            | |
                       SFI interlock
```

Here, the value being modified cannot be referenced as an operand until after it has been written, causing a delay of 2 cycles. The solution to this interlock is similar to the others, but this one requires 2 instructions inserted between the STore and the Add:

```
ST    R2,ABC         G B L E W
L     R3,LOC1          G B L E W
L     R4,LOC2            G B L E W
A     R1,ABC               G B L E W
```

There is a better solution to this particular sequence, however, and that is to replace the Add instruction with an AR (Add Register) instruction:

```
ST    R2,ABC        G B L E W
AR    R1,R2           G B L E W
```

Now, there are no interlocks, and no additional instructions are required. This example is not as specious as it at first appears. In most cases of SFI interlocks, the value being stored is being stored from a register, and it can be used from the source as well as the destination in many instances.

There are other classes of interlocks, but those are the most common ones that it is feasible for the programmer to eliminate. Of course, the cost of rearranging code as suggested here is in terms of readability, and thus, maintainability. It is also the case that for these techniques to be successful, the instruction sequence must contain a number of unrelated instructions. Typically, logic constraints prevent moving instructions around enough to eliminate all interlocks.

Another phenomenon which occurs when moving instructions around is that one interlock will be removed, only to be replaced by another. The existence of bypasses (see below) complicates the pipe's behavior to the point that it is very difficult to predict the result of a substantial change to an instruction sequence.

4.3 Pipeline bypasses

Pipelined processors usually include bypasses from stage to stage within the pipe to avoid interlocks. The EEI example above illustrates one in the 580, which is why the interlock case has only a 1 cycle delay instead of a 2 cycle delay. Another commonly used bypass in the 580 pipeline improves the flow of a Load-STore sequence. If there were no bypasses in the pipeline, a Load-STore sequence would look like this:

```
L     R1,LOC1       G B L E W
ST    R1,LOC2         G B B B B L E W
```

In fact, there is a bypass here which is a special data path that allows the result from the E-cycle in the Load instruction to be fed back to the beginning of the E-cycle of the STore instruction. So the Load-STore sequence looks like this:

```
L     R1,LOC1       G B L E W
ST    R1,LOC2         G B L E W
```

This bypass is effective for any instruction following the Load instruction which uses the contents of the register as an operand — i.e. this eliminates an EEI-type interlock. It was found that using this bypass could eliminate a large number of interlocks in the routines that were optimized here. For these sequences, the instructions could either be separated by several instructions, or put immediately adjacent.

An interesting consequence of bypasses is that some instruction sequence modifications which appear to be improvements in fact introduce delays instead of reducing them. As an example, consider the following sequence:

```
L     R1,LOC1       G B L E W
ST    R1,LOC3         G B L E W
L     R2,LOC2           G B L E W
```

Without using the processor simulator, it would appear that the sequence would be better if it was rearranged as follows:

```
L     R1,LOC1       G B L E W
L     R2,LOC2         G B L E W
ST    R1,LOC3           G B B B L E W
                          | |
                        EEI interlock
```

In fact, as can be seen from the picture of the pipe flow, the reordered sequence is worse.

Another example of the complications which bypasses introduce into pipeline optimization is as follows. This is a sequence which both uses an operand and copies it to another storage location. It is not obvious that there should be a difference in the pipeline characteristics of the two alternative sequences. The first sequence uses the bypass between the L (Load) and AR (Add Register) instructions, but defeats the bypass between the L and ST (STore) instructions.

```
L     R1,LOC1       G B L E W
AR    R2,R1           G B L E W
ST    R1,LOC2           G B B B L E W
                          | |
                        EEI interlock
```

The second alternative uses the bypass between the L and ST. However, the AR instruction does not require a bypass because the value in R1 has been updated in time (the L cycle).

```
L     R1,LOC1       G B L E W
ST    R1,LOC2         G B L E W
AR    R2,R1             G B L E W
```

4.4 High-speed buffer considerations

Buffer (cache) misses could have a significant impact on the execution time of a sequence. If an instruction or operand is not in the buffer, a penalty of anywhere from 11 to 30 cycles could occur, depending on the special case. Just one or two buffer misses could lengthen an instruction sequence by up to 30%. For instructions, the execution frequency of the path will determine the likelihood that the path suffers buffer misses. For operands, the reference frequency will determine the miss frequency. Consequently, if paths overlap (i.e. use common subroutines) or use common data areas, the reference frequencies of those common items will be higher than they would be otherwise, and therefore, the path's overall miss rate will go down.

The high-speed buffer in the 580 processor is managed in a fairly conventional way, approximating a least-recently used replacement policy. Depending on the workload's memory reference characteristics, the frequency of reference required to maintain a given memory location in the buffer can be calculated. This frequency was found to be around 600 invocations per second for the workloads studied. So, those paths which were executed less frequently than 600 times per second were candidates for use of subroutines for common operations, while those with a higher frequency always used inline code.

It turned out that the most common paths had invocation frequencies greater than 600 per second, so using inline code was always better than using subroutines. However, there were paths whose frequencies were not high enough to assure buffer residence. In these cases, it was found that by using a much more special purpose (i.e. ad hoc) calling convention, the space and buffer benefits of using a subroutine could be obtained at an acceptable cost in terms of additional cycles in the path.

Most of these routines use a work area for scratch storage. By ensuring that they all use the same work area, that set of memory locations is guaranteed to be referenced frequently enough to always be buffer-resident. Similarly, most routines use assembled-in constants. By packing all the constants together, the minimum number of buffer locations can be used, and their frequency will be high enough to assure buffer-residence.

4.5 Hardware modification

Changing the hardware was an option which was available to this project that is not normally available to a software development effort. In this case, two expensive instructions were improved (substantially), resulting in an overall reduction of 12% in the weighted average path length for these routines. Because changing the hardware is the most expensive optimization available, it is very difficult to make a case for a hardware change unless all avenues for improving the software have been explored. Once the frequent paths had been analyzed by the processor simulator, the long instructions became obvious, and the impact of speeding them up could be determined easily.

5.0 Conclusions

The techniques used in this project are well-established, with the exception of the pipeline optimizations. The highlights of this effort lie in the tools used. Obtaining the path frequencies is the crucial first step of an organized optimization effort. Obtaining path execution characteristics is likewise necessary, though the method used here is not the only one that could be used. Particularly, an estimate of path timings could have been obtained using graph techniques as described in [9].

The results of doing these optimizations were significant. As in most performance enhancement projects, a few changes were made at a time, with measurement after each set of changes. As a result, the performance enhancement due to each set of changes could be tracked, allowing the effectiveness of these techniques to be evaluated, to the extent that each set of changes implements a different optimization method. In fact, some of the changes represented application of a single optimization, and some represented more than one. From the data obtained, we can infer, approximately, the effectiveness of each technique.

Overall, the weighted average execution time of the paths in question was reduced by 56% for the batch workload, 59% for the time-sharing workload, and 61% for the transaction-processing workload. The improvement due to the different optimizations is (approximately) as follows:

	% improvement		
optimization	batch	TSO	TP
subroutine calls	19%	21%	7%
common cases and branching	12%	13%	10%
hardware	12%	11%	24%
instruction sequences	10%	10%	10%
buffer considerations	3%	3%	10%
total	56%	58%	61%

The optimization techniques themselves are generally applicable to other software improvement projects. Optimizing common cases, common paths, and subroutine usage are all general techniques that can be applied with little more than knowledge of path frequencies required. Optimizing expensive instructions and sequences requires detailed knowledge of the pipeline structure of the processor. This would be very difficult without a tool like the processor simulator.

Acknowledgements

The code optimizations described here were done by John Andoh, Joe Tuminaro, and the author. Mark Goheen wrote the pipeline simulator and helped to analyze code sequences. Dave Stinchcomb supplied the early frequency estimates and facilitated use of the subject workloads. Mac MacDougall also contributed to the success of the project.

References

1. Bentley, Jon, Writing Efficient Programs. Prentice-Hall, New Jersey, 1982.

2. DeMillo, R.A., S.C. Eisenstat, and R.J. Lipton, "Can Structured Programs be Efficient?", Sigplan Notices, Vol. 11, No. 10, Oct. 1976.

3. Dongarra, J.J. and A.R. Hinds, "Unrolling Loops in Fortran", Software Practice and Experience, Vol. 9, No. 3, March 1979.

4. Fisher, J.A. and J.J. O'Donnell, "VLIW Machines: Multiprocessors We Can Actually Program", Spring CompCon 84 Digest of Papers, Feb. 1984.

5. Hennessy, John and Thomas Gross, "Postpass Code Optimization of Pipeline Constraints," Transactions on Programming Languages and Systems, Vol. 5, No. 3, July 1983.

6. MacDougall, M.H. "Instruction-Level Model of Performance for Processor Design," IEEE Computer, Vol. 17, No. 7, July 1984.

7. Ramamoorthy, C.V. and H.F. Li, "Pipeline Architecture". Computing Surveys, March 1977, pp.61-102.

8. Rymarczyk, James W., "Coding Guidelines for Pipelined Processors," Computer Architecture News, Vol. 10, No. 2, March 1982, pp.12-19.

9. Smith, C.U. and J.C. Browne, "Performance Engineering of Software Systems: A Case Study." Proc. AFIPS National Computer Conference, Houston, June 1982.