

The influence of hardware on software optimization

Alberto Villarreal

ABSTRACT

One of the key factors determining computer performance is the degree to which the hardware can take advantage of *Instruction Level Parallelism* (ILP) that software often exhibits. ILP refers to the parallelism between individual operations in a program, and it is exploited by hardware elements called *pipelines* that are able to execute multiple instructions concurrently. Research in this area has been reported extensively in the literature; however, much of it contains technical language and is not addressed to the final user. In this document, I describe in simple terms how modern processors are able to deliver high performance, and how ILP in programs can be exploited using pipelining and other hardware elements (*cache memories* and *branch predictors*) to increase the delivered performance of a processor.

Key words: software optimization, pipelining, cache memory, branch prediction

Introduction

When running computer programs, one computer frequently will deliver better results than another, even if both computers have processors with the same clock speed, for example, a 75 MHz Pentium processor compared to a 75 Mhz MIPS R8000 processor. Also, incrementing the clock speed of a processor may not increment the performance of the program proportionately. In some cases, it may not increment at all. The main reason for this is that processors commonly used in workstations today incorporate supercomputer technology to a certain degree, such as pipelining and other hardware-implemented parallel techniques. Note that here I am not referring to the coarse-grained parallelism between large sets of operations, such as subprograms, which is exploited by multiprocessor systems. Rather, I am referring to a fine-grain one called *Instruction level parallelism* (ILP) which refers to the parallelism between individual operations that can be exploited by uniprocessor systems.

If the software does not take advantage of ILP processing, the delivered performance will be less than optimal (Franklin, 1993). On the other hand, attention paid to a few key optimization procedures can pay off with important increases in performance. Recent studies have confirmed that a large amount of ILP exists in ordinary

numerical programs. This parallelism can be exploited by the hardware (Franklin, 1993), (Wall, 1993). This means that no matter how much parallelism is exploited by coarse-grain parallel processors, a substantial amount of parallelism will still remain to be exploited at the instruction level. Therefore, irrespective of the speedup given by coarse-grain parallelism, ILP processing can increase that speedup substantially.

Hardware is not the only issue involved in software optimization. In the past few years, optimizing compilers have become an essential component of modern high-performance computer systems. Besides generating executable code, these compilers analyze the user's source code and apply various transformations to improve its performance by exploiting ILP present in the software (Bacon *et al.*, 1994). However, these compilers are not always effective when analyzing complex code (Saavedra & Smith, 1992). In particular, some widely used compilers, such as the freely available GNU's *gcc* and *g77*, have limited code analysis and code transformation capabilities.

On the other hand, modifying existing codes (or designing new ones) to exploit ILP, independently of the compiler, presents the following advantages:

- The performance of the software will be more pre-

dictable in terms of hardware parameters, such as depth of the pipelines, amount of cache memory available, etc.

- The program will be able to take advantage of hardware optimization mechanisms, such as hardware-based branch prediction and out-of-order execution, irrespective of the compiler being used; Thus, the performance of non-expensive and broadly available computer systems using public domain software, such as Pentium-based systems currently installed at CWP, may be able to increase substantially and to approach that of more expensive systems.

- If an efficient optimizing compiler is available, it will be able to extract the best performance of an already optimized code. This would be achieved without performing extensive and time-consuming analysis and transformations that could affect the floating point accuracy of the program.

- Obtaining the best performance in a uniprocessor system is critical when the code is to take advantage of coarse-grain multiprocessor parallelism. If the program is to be executed in parallel in a heterogeneous environment, where different compilers will generate the code for each architecture present in the distributed system, caution must be taken to insure that the part of the code at each node is running fast in order not to decrease the overall performance of the distributed system.

Understanding something about crucial hardware and software elements built into modern computer systems is extremely useful to increasing the delivered performance of computation-intensive programs.

In this document I describe in simple terms some of the hardware elements present in modern processors, such as pipelines, branch predictors and cache memories, and how those elements can take advantage of the ILP present in software to reduce the execution time. I also discuss how taking these elements into account when coding can often increase the speed of a program to an important fraction of the processor's peak speed.

A few words on modern computer architecture

High performance in modern computer systems is achieved by incrementing the speed when performing arithmetic operations, as well as incrementing the speed of the memory delivering data for those arithmetic operations to be completed. In this section I briefly describe how modern processors improve those two parameters using *pipelines*, *cache memories*, and *branch prediction mechanisms*. These processors are designed to exploit the parallelism that programs exhibit at the instruction

level. As an example of this parallelism, let us look at the following code fragment:

```
i1 = array[j];
i2 = i3 * N;
array[j+1] = i3;
```

This code fragment consists of three instructions that can be executed concurrently, because they have no data dependencies, i.e., they do not depend on each other's results. The following code, on the other hand, does have data dependencies and cannot be executed in parallel:

```
i1 = array[j];
i2 = i1 * N;
array[i2] = i3;
```

Much of the work in scalar optimization is aimed at eliminating or reducing data dependencies in programs in order to increase the ILP and permit the hardware to efficiently execute the instructions.

Pipelines, together with *cache memories* and *branch prediction systems*, are some of the most important components of *superscalar* processors. A superscalar processor is one that can issue multiple independent instructions in the same cycle. Superscalar processing has been acclaimed as "vector processing for scalar programs," and speedup factors between 4 and 10, depending on the amount of ILP present in the test codes, have been reported when using this technique (Wall, 1993). Descriptions of the superscalar architecture of some common processors can be found in Zagha (1996), Hunt (1995), and the Pentium Pro and Pentium II pages on Intel's site^{*}.

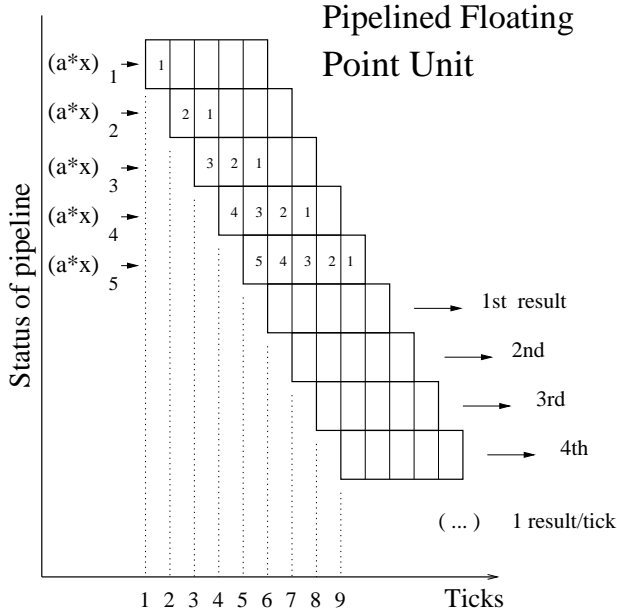
Note, however, that other hardware issues such as networking, hard disk speed, graphical accelerators, etc., also have important influence on the delivered performance of the system. Here I will only talk about some in-processor components that affect performance.

Pipelines

Pipelining is the most common implementation technique used today to increase the performance of processors. Pipelines in modern processors replace sequential arithmetic units present in old hardware. Their purpose is to reduce the number of processor *ticks*, or clock cycles, needed to execute an individual instruction, by taking advantage of ILP found in everyday programs.

The idea behind pipelining is that if the hardware unit performing the arithmetic operations (the pipeline)

^{*} <http://www.intel.com>



Sequential mode: 5 ticks/instr.
 Pipelined mode: 1 ticks/instr.

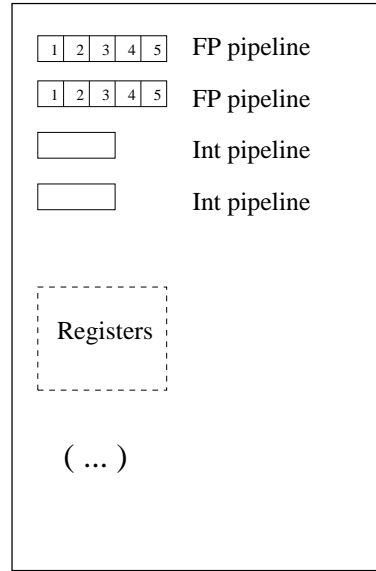
Figure 1. Flow of instructions through a pipeline. In this 5-stage pipeline, 1 instruction takes 5 ticks to complete, but a new instruction is started every tick, and 1 instruction is completed in each tick.

is segmented in several stages, and each stage executes pieces of an operation concurrently and independently of one another, then it is possible to start another operation just after the previous one completes the first stage of the pipeline, providing operations are of the same type (Dowd, 1993). A pipeline can be thought of an assembly line where multiple actions (instructions, in this case) are performed concurrently.

Figure (1) shows schematically how a 5-stage floating point pipeline works. Multiplication operations enter the pipeline and at every cycle, the pipeline rolls them over until, after 5 cycles, the first result is popped out. After that, because the pipeline stages are independent of one another, up to five operations can be in the pipeline at a time, and the pipeline will produce a result every clock cycle as long as input operations exist. This is a powerful mechanism: where before it would have taken five ticks for a single result to come out, now the pipeline generates as much as one result every tick.

As an example of how the pipeline affects the peak performance of processors, let us look at a simplified representation of the structure of a typical 75 Mhz processor as shown in Figure (2).

This processor contains 2 pipelined floating point



(2 flop) (75 Mhz) = 150 Mflops

Figure 2. Schematic structure of a processor showing the structure of pipelines. This particular processor has two Floating point pipelines (FP) and two integer pipelines (Int). Floating point pipelines can perform Floating point additions or Floating point multiplications.

units and 2 pipelined integer units. When working in fully pipelined mode, each of the floating point units can perform up to 1 floating point operation per tick, as described in Figure (1). If both floating point units are working simultaneously, then the peak floating point performance of this processor is 2 flops/tick, which at a clock speed of 75 MHz, corresponds to a peak performance of 150 Mflops.

In general, a pipeline with S stages working in fully pipelined mode can execute N operations in

$$T_p = S + (N - 1) \tag{1}$$

ticks. And we can observe that as the number of operations increases, the speed of the pipeline approaches the one operation per tick rate.

However, if the pipeline is working in sequential mode, i.e., starting the next operation once the previous one has been fully completed, then the same N operations will be executed in

$$T_s = NS \tag{2}$$

ticks. From this we see that the ideal pipeline speedup:

$$T_s/T_p \tag{3}$$

equals the number of stages S when N is large, assuming there is no overhead involved in the pipeline mechanism).

This explains why processors with longer pipelines increment the theoretical (peak) performance. Note, however, that the longer the pipeline, the bigger the penalty when the pipeline fails to work in fully pipelined mode, so another solution to keep improving the performance, without increasing the clock speed or the number of stages, is to add extra pipeline units to the processor, as in the case of the processor shown in Figure (2), which has 2 floating point pipelines and 2 integer pipelines.

Every instruction in a processor, floating point addition, subtraction, and multiplication, as well as integer operations and memory references, can be pipelined. However, in order for this technique to work, the code must be as close as possible to a stream of operations with no branches, i.e., no *if*'s, *go to*'s, subroutine calls, etc. The reason is that every time a branch is encountered the pipeline will stop the flow of operations, reset, and start again with a new flow of operations, and this will result in a big penalty in processing time.

Branch prediction and speculative execution

Unfortunately, most programs make heavy use of branches, and both the compiler and the hardware must have mechanisms to cope with control transfers in the code.

How does the pipeline know when a branch is coming? There are several mechanisms for this. In some cases an optimizing compiler will try to predict which way it believes a branch is likely to go, and it will generate appropriate object code trying to use as much of the pipeline capabilities as it can. In other cases, hardware mechanisms working at execution time permit the CPU to guess where a branch is going to appear based on when other branches have occurred (Dowd, 1993). In either case, the more we simplify the branches in the code, the more efficient these mechanisms will work, and the more efficient the pipeline will behave.

The worst scenario (unfortunately a very common one when developing complex software) is when every iteration in a loop depends on a branch decision (an *if* inside a loop). If this scenario cannot be transformed by the compiler or the hardware, the processor will be working in sequential mode at a very low speed. Fortunately, recent research in algorithms show that most of these kinds of inefficient structures can be transformed into others that can be fully, or at least partially, pipelined. For example consider the following commonly used structure, known as *invariant test*:

```
DO I=1,K
  IF (N .EQ. 0) THEN
    A(I) = A(I) + B(I) * C
  ELSE
    A(I) = 0.
  ENDIF
ENDDO
```

“Invariant” in this case means that the result of the test will not depend on the result of the operations in the loop. The value of N will be the same regardless of the values of the variables A , B and C . Therefore, the loop can be rewritten making the test outside the loop and replicating the loop body twice: once for a valid outcome, and once for a false outcome:

```
IF (N .EQ. 0) THEN
  DO I=1,K
    A(I) = A(I) + B(I) * C
  ENDDO
ELSE
  DO I=1,K
    A(I) = 0.
  ENDDO
ENDIF
```

This is one of the simplest transformations in loops, yet the effect on the run time can be important, because now the computations inside the loop do not depend on the if-statement, and are, therefore, much easier to pipeline. Besides, $N - 1$ instances of the test have been eliminated. Although most optimizing compilers will recognize this structure and will make the appropriate transformations in order for the loop to exhibit ILP and be processed in superscalar mode, it is a good practice to make the transformations manually. Code written in this way will be able to run efficiently independently of the compiler.

Another frequently used structure is the so called *Dependent loop conditional*:

```
DO I=1,K
  IF (X .LT. A(I)) THEN
    X = X + B(I)
  ELSE
    X = X + C(I)
  ENDIF
ENDDO
```

This loop has dependencies between iterations, which means that it is not possible to know which way the branch will go for the next iteration until the current one is done. This is a less trivial case than the previous Invariant Test and the compiler may not be able

to optimize it. The way the hardware can cope with this kind of structures is by trying to estimate which is the most probable result of the test in the if-statement using *branch prediction* systems. These systems try to predict the way the instruction will branch the next times it is used.

Research has been done on how to increase parallelism on conditionals as the one shown above using branch prediction. Several authors have suggested ways of predicting the direction of conditional branches with hardware and/or software that uses the history of previous branches. Different mechanisms take advantage of different observed patterns in branch behavior. Descriptions and comparisons of different branch prediction mechanisms can be found in Wall (1994),(1993) and McFarling (1993). One of the most recent methods that has been implemented in actual processors is the *Dynamic Branch Prediction* mechanism, which looks multiple steps ahead in the program and makes a prediction for the direction of the following branches based on the direction the branch went the last times it was executed. More accurate predictions can be made by utilizing more branch history. Prediction accuracy of 90 - 97% has been reported using these systems (Wall, 1994), and they perform well either when each branch is strongly biased in a particular direction, for example, when the condition is almost always false in the loop shown above, or for branches with simple repetitive patterns.

Once the processor has predicted the direction of the upcoming branches, it creates a schedule of instructions to be executed in an optimal way by the pipelines, trying to use its full superscalar power, and executes it “speculatively,” storing the results of these instructions as “speculative results” until the final state can be determined, when the branches are resolved. Once this happens, the temporary results can be either copied to the registers, if the prediction of the branch was successful, or discarded and the pipeline flushed, if the branch was incorrectly predicted. This process is called *Speculative execution*, and if the processor is also capable of scheduling the instructions in a different order than the original order implied by the program, to avoid data dependencies, for example, then the process is called *Out-of-order execution*. In this case, the instructions will be returned to their proper program order once the branch direction is resolved.

This is a powerful mechanism for exploiting ILP masked by conditional branches. And most modern workstations are based on processors using some implementation of branch prediction and speculative execution (MIPS, Alpha, HP-PA, Power, Pentium Pro, among others).

Regarding the example loops shown above, the compiler may be able to rearrange the conditional by itself (more likely in the first example than in the second one), but if the software is written following basic optimization techniques to increase ILP, then the code will perform well on any machine, even on those that do not have smart compilers. The second example shows how the software can take advantage of hardware-implemented optimization capabilities, such as branch prediction systems and speculative execution techniques. Optimizing these structures via software requires more programmer-input and, therefore, the compiler will be rather conservative with these kinds of transformations. For this case, the conditional can be arranged manually in such a way that the condition is either true or false for a large number of consecutive iterations (i.e., making the outcome of the branch more predictable), increasing this way the chances that the processor successfully predicts the outcome of the conditional.

These and other examples are well referenced in the literature, for example in Dowd (1993) and in Bacon (1994).

Cache memory

Almost any modern computer system has two types of system memory: main memory and cache memory. The cache memory is a (usually) small, high speed memory that contains the most recently accessed pieces of main memory (Figure (3)). This high speed memory is necessary in modern systems because the times it takes to bring data into the processor from main memory is long compared to the time it takes to execute an instruction, and this performance gap between processors and memory is widening; for example, in a Pentium Pro-based system, the access time for main memory is about 60 nanoseconds (ns), while a 100 MHz processor can execute most instructions in 1 clock or 10 ns. Therefore, a bottleneck is formed at the input of the processor. A typical access time for cache memory in this systems is about 15 ns. As a result, cache memory improves the efficiency of the system allowing small portions of data in main memory to be accessed about 4 times faster.

Since pipelines are trying to execute an operation every tick and, therefore, loading and storing data every tick, then the software will not be able to exploit pipelining power by using only main memory accesses, even if the software presents a high degree of ILP. For this reason, the efficiency of the memory system must be increased in order to take advantage of superscalar processing. The cache memory improves efficiency using the concept of “data locality” (spatial or temporal). “Locality” refers to a characteristic that programs often ex-

hibit, in which they very often make use of data which are near in memory, both in space and time, to other pieces of data which are going to be used immediately. More specifically, we can say that *Temporal Locality* implies that, when a data element is referenced in a program, it will be referenced again soon, and *Spatial Locality* implies that when a data element is referenced, nearby data elements will be referenced soon (Dowd, 1993). Under this scenario, chances are good that the next time the program needs data, these data will be ready in cache memory for fast access, and the CPU will not have to perform the slow process of retrieving it from main memory.

Therefore, a good way to improve the performance of the program is to improve data locality (Saavedra & Smith, 1993), which can be performed in several ways. Again, an optimizing compiler will try to transform the code in order to exploit locality, but this will not always occur, especially in complex code. Manually organizing the loops in the program so that contiguous operations use contiguous pieces of data in memory is the most effective way to improve cache memory use and, therefore, to increase pipelining use.

A common example of cache memory optimization is *Blocking memory references* (see for example Dowd(1993)). This example is common in geophysical software where large multidimensional arrays must be accessed in memory, and loop transformations are necessary to improve locality. The following loop:

```
DO I=1,N
  DO J=1,N
    DO K=1,N

      C(I,J)=C(I,J)+A(I,K)*B(K,J)

    END DO
  END DO
END DO
```

has a big stride in some matrix (matrix **B** if FORTRAN is used, or matrix **A** if **C** is used), so there will be cache misses, i.e., some data will not be available in cache when the operation is ready to execute in the pipeline, and a slow main memory access will be necessary. This can be avoided transforming the loop to operate in submatrices that can fit in cache:

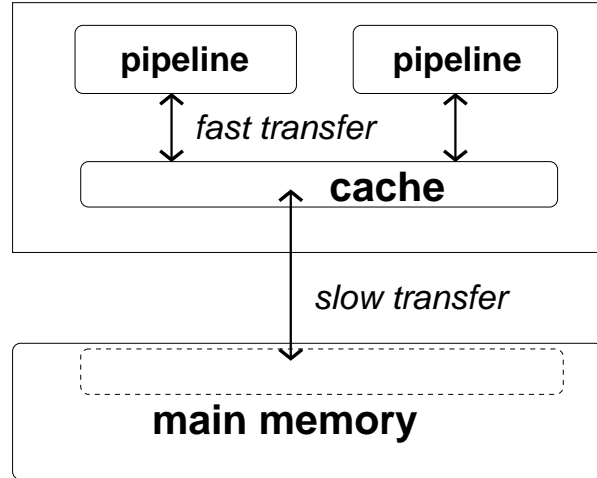


Figure 3. Memory hierarchy in a computer system. Cache memory is a small, fast-access memory which stores chunks of data transferred from/to main memory and keeps data used most often.

```
DO IB=1,N,NB
  DO JB=1,N,NB
    DO KB=1,N,NB

      DO I=IB,MIN(N,IB+NB-1)
        DO J=JB,MIN(N,JB+NB-1)
          DO K=KB,MIN(N,KB+NB-1)

            C(I,J)=C(I,J)+A(I,K)*B(K,J)

          END DO
        END DO
      END DO

    END DO
  END DO
END DO
```

where **NB** is the number of submatrices the original matrix is partitioned into. The size of the submatrices must be such that the three submatrices (from **A**, **B** and **C**) fit into cache memory. We can estimate the size of these submatrices considering that if each one in this example is $n \times n \times n$, and each one contains elements of type REAL*S (in FORTRAN nomenclature), the biggest n such that all three submatrices fit completely in cache must be

$$n < \text{extra} + \sqrt[3]{\frac{\text{cache} - \text{size}}{3S}}, \quad (4)$$

where *extra* is extra space in cache that will be required by other variables and *cache - size* is the total cache size in bytes.

Remarks

Code transformation for best performance in uniprocessor systems (also called scalar optimization) is an important issue in modern high performance processors. Scalar optimization techniques are aimed basically at uncovering the ILP present in software.

Most of the CPU time consumed by a program developed for a scientific application is spent on loop-based manipulation of arrays, so the analysis of a program can be restricted only to the most time-consuming loops contained in it. Optimizing transformations in loops and conditionals is an active area of current research, and much of this research is oriented toward developing better optimizing compilers. However, some of these transformations and techniques can be applied manually when developing or updating software for high-performance processors. From my experience, in most cases only a few transformations that improve the locality and, hence, the ILP of the code, will cause important improvement in the performance of the program, because the locality will permit pipelines and branch predictors to work together providing superscalar performance.

In summary, exploiting the ILP of software to get maximum efficiency from a processor has several important implications when developing software:

- Often, expensive hardware can be substituted by low-cost workstations when the programs are optimized, because the peak performance of expensive processors is often not much higher than that of moderate-cost processors.
- It makes little sense to compare different algorithms using only the processing time it takes for the code to run, even in the same computer, because different algorithms will present different degrees of ILP. Therefore, besides the number of operations itself, the structure of the algorithm becomes another metric for algorithm comparison. In particular, one characteristic in algorithms called “locality” is one of the main factors affecting the performance. Poor locality in algorithms will decrease almost any opportunity for optimization. I have discussed locality and the way it can take advantage of cache memories, branch predictors, and pipelines in order to obtain maximum performance from software.
- Obtaining optimum performance in scalar mode, i.e., in one processor, is critical before the coarse-grained parallelization is done. If the program is making inefficient use of processor and memory resources, this inefficiency will be multiplied when going parallel and the overall efficiency of the system will decrease.

Extensive surveys of optimizing transformations can

be found in the literature, for example, in Bacon *et al.*, (1994).

References

- Bacon, D. F., Graham, S. L., & Sharp, O. J. 1994. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, **26**(4), 345–419.
- Dowd, Kevin. 1993. *High Performance Computing*. O'Really.
- Franklin, M. 1993. *The multiscalar architecture*. Ph.D. thesis, University of Wisconsin - Madison.
- Hunt, D. 1995. Advanced performance features of the 64-bit PA-8000. *In: COMPCON'95*.
- McFarling, S. 1993 (June). *Combining branch predictors*. Tech. rept. Technical Note TN-36. Digital Western Research Laboratory.
- Saavedra, R. H., & Smith, A. J. 1992 (August). *Performance characterization of optimizing compilers*. Tech. rept. USC-CS-92-525. Dept. of Computer Science, University of Southern California.
- Saavedra, R. H., & Smith, A. J. 1993 (July). *Measuring cache and TLB performance and their effect on benchmark runt times*. Tech. rept. USC-CS-93-546. Dept. of Computer Science, University of Southern California.
- Wall, D.W. 1993 (November). *Limits of instruction level parallelism*. Tech. rept. Research Report 93/6. Digital Western Research Laboratory.
- Wall, D.W. 1994 (March). *Speculative execution and instruction-level parallelism*. Tech. rept. Technical Note TN-42. Digital Western Research Laboratory.
- Zagha, M., Larson, B, Turner, S., & Itzkowitz, M. 1996. Performance analysis using the MIPS R10000 performance counters. *In: Supercomputing 96*.

