Exploiting Instruction Level Parallelism with Software Approaches

1Q. Explain basic compile Techniques for Exposing ILP

4.1 Basic Compiler Techniques for Exposing ILP

Basic Pipeline Scheduling and Loop Unrolling

To avoid a pipeline stall, a dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of that source instruction. A compiler's ability to perform this scheduling depends both on the amount of ILP available in the program and on the latencies of the functional units in the pipeline. Throughout this chapter we will assume the FP unit latencies shown in Figure 4.1,

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

FIGURE 4.1 Latencies of FP operations used in this chapter. The first column shows the originating instruction type. The second column is the type of the consuming instruction. The last column is the number of intervening clock cycles needed to avoid a stall.

We will rely on an example similar to the one we used in the last chapter, adding a scalar to a vector:

```
for (i=1000; i>0; i=i–1)
        x[i] = x[i] + s;
```

This loop is parallel by noticing that the body of each iteration is independent. The first step is to translate the above segment to MIPS assembly language. In the following code segment, R1is initially the address of the element in the array with the highest address, and F2 contains the scalar value, s. Register R2 is precomputed, so that 8(R2) is the last

element to operate on. The straightforward MIPS code, not scheduled for the pipeline, looks like this:

```
Loop:  L.D      F0,0(R1)     ;F0=array element
       ADD.D    F4,F0,F2      ;add scalar in F2
       S.D      F4,0(R1)      ;store result
       DADDUI   R1,R1,#-8     ;decrement pointer
                              ;8 bytes (per DW)
       BNE      R1,R2,Loop    ;branch R1!=zero
```

Let's start by seeing how well this loop will run when it is scheduled on a simple pipeline for MIPS with the latencies from Figure 4.1.

```
                                  Clock cycle issued
Loop:  L.D      F0,0(R1)          1
       Stall                      2
       ADD.D    F4,F0,F2          3
       stall                      4
       stall                      5
       S.D      F4,0(R1)          6
       DADDUI   R1,R1,#-8         7
       stall                      8
       BNE      R1,R2,Loop        9
       stall                     10
```

This code requires 10 clock cycles per iteration. We can schedule the loop to obtain only one stall:

```
Loop:  L.D      F0,0(R1)
       DADDUI   R1,R1,#-8
       ADD.D    F4,F0,F2
       stall
       BNE      R1,R2,Loop    ;delayed branch
       S.D      F4,8(R1)      ;altered & interchanged with DADDUI
```

Execution time has been reduced from 10 clock cycles to 6.The stall after ADD.D is for the use by the S.D.

In the above example, we complete one loop iteration and store back one array element every 6 clock cycles, but the actual work of operating on the array element takes just 3 (the load, add, and store) of those 6 clock cycles. The remaining 3 clock cycles consist of loop overhead—the DADDUI and BNE—and a stall. To eliminate these 3 clock cycles we need to get more operations within the loop relative to the number of overhead instructions.

A simple scheme for increasing the number of instructions relative to the branch and overhead instructions is *loop unrolling*. Unrolling simply replicates the loop body multiple times, adjusting the loop termination code.

Loop unrolling can also be used to improve scheduling. Because it eliminates the branch, it allows instructions from different iterations to be scheduled together. In this case, we can eliminate the data use stall by creating additional independent instructions within the loop body. If we simply replicated the instructions when we unrolled the loop, the resulting use of the same registers could prevent us from effectively scheduling the loop. Thus, we will want to use different registers for each iteration, increasing the required register count.

In real programs we do not usually know the upper bound on the loop. Suppose it is *n*, and we would like to unroll the loop to make *k* copies of the body. Instead of a single unrolled loop, we generate a pair of consecutive loops. The first executes (*n* mod *k*) times and has a body that is the original loop. The second is the unrolled body surrounded by an outer loop that iterates (*n*/*k*) times. For large values of *n*, most of the execution time will be spent in the unrolled loop body.

Summary of the Loop Unrolling and Scheduling Example

To obtain the final unrolled code we had to make the following decisions and transformations:

1 Determine that it was legal to move the S.D after the DADDUI and BNE, and find the amount to adjust the S.D offset.

2 Determine that unrolling the loop would be useful by finding that the loop iterations were independent, except for the loop maintenance code.

3 Use different registers to avoid unnecessary constraints that would be forced by using the same registers for different computations.

4 Eliminate the extra test and branch instructions and adjust the loop termination and iteration code.

5 Determine that the loads and stores in the unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent. This transformation requires analyzing the memory addresses and finding that they do not refer to the same address.

6. Schedule the code, preserving any dependences needed to yield the same result as the original code.

The key requirement underlying all of these transformations is an understanding of how an instruction depends on another and how the instructions can be changed or reordered given the dependences.

There are three different types of limits to the gains that can be achieved by loop unrolling: a decrease in the amount of overhead amortized with each unroll, code size limitations, and compiler limitations. Let's consider the question of loop overhead first. When we unrolled the loop four times, it generated sufficient parallelism among the instructions that the loop could be scheduled with no stall cycles. A second limit to unrolling is the growth in code size that results.

2Q. Explain how to reduce branching cost by Static Branch Prediction method

4.2 Static Branch Prediction

Static branch predictors are used in processors where the expectation is that branch behavior is highly predictable at compile-time; static prediction can also be used to assist dynamic predictors.

An architectural feature that supports static branch predication, namely delayed branches. Delayed branches expose a pipeline hazard so that the compiler can reduce the penalty associated with the hazard. The effectiveness of this technique partly depends on whether we correctly guess which way a branch will go. Being able to accurately predict a branch at compile time is also helpful for scheduling data hazards. Loop unrolling is on simple example of this; another example, arises from conditional selection branches. Consider the following code segment:

```
     LD      R1,0(R2)
     DSUBU   R1,R1,R3
     BEQZ    R1,L
     OR      R4,R5,R6
     DADDU   R10,R4,R3
L:   DADDU   R7,R8,R9
```

The dependence of the DSUBU and BEQZ on the LD instruction means that a stall will be needed after the LD. Suppose this branch was almost always taken and that

the value of R7 was not needed on the fall-through path. Then we could increase the speed of the program by moving the instruction DADD R7,R8,R9 to the position after the LD. If the branch was rarely taken and that the value of R4 was not needed on the taken path, then we could contemplate moving the OR instruction after the LD.

To perform these optimizations, we need to predict the branch statically when we compile the program. There are several different methods to statically predict branch behavior. The simplest scheme is to predict a branch as taken. This scheme has an average misprediction rate that is equal to the untaken branch frequency, which for the SPEC programs is 34%. Unfortunately, the misprediction rate ranges from not very accurate (59%) to highly accurate (9%).

A better alternative is to predict on the basis of branch direction, choosing backward-going branches to be taken and forward-going branches to be not taken. For some programs and compilation systems, the frequency of forward taken branches may be significantly less than 50%, and this scheme will do better than just predicting all branches as taken. In the SPEC programs, however, more than half of the forward-going branches are taken. Hence, predicting all branches as taken is the better approach.

A still more accurate technique is to predict branches on the basis of profile information collected from earlier runs. The key observation that makes this worthwhile is that the behavior of branches is often bimodally distributed; that is, an individual branch is often highly biased toward taken or untaken.

3Q. Explain VLIW Approach to avoid dependences.

4.3 Static Multiple Issue : The VLIW Approach

The compiler may be required to ensure that dependences within the issue packet cannot be present or, at a minimum, indicate when a dependence may be present.

The first multiple-issue processors that required the instruction stream to be explicitly organized to avoid dependences. This architectural approach was named VLIW, standing for Very Long Instruction Word, and denoting that the instructions, since they contained several instructions, were very wide (64 to 128 bits, or more). The basic architectural concepts and compiler technology are the same whether multiple operations are

organized into a single instruction, or whether a set of instructions in an issue packet is preconfigured by a compiler to exclude dependent operations (since the issue packet can be thought of as a very large instruction). Early VLIWs were quite rigid in their instruction formats and effectively required recompilation of programs for different versions of the hardware.

The Basic VLIW Approach

VLIWs use multiple, independent functional units. Rather than attempting to issue multiple, independent instructions to the units, a VLIW packages the multiple operations into one very long instruction, or requires that the instructions in the issue packet satisfy the same constraints. we will assume that multiple operations are placed in one instruction, as in the original VLIW approach. Since the burden for choosing the instructions to be issued simultaneously falls on the compiler, the hardware in a superscalar to make these issue decisions is unneeded.

Since this advantage of a VLIW increases as the maximum issue rate grows, we focus on a wider-issue processor. Indeed, for simple two issue processors, the overhead of a superscalar is probably minimal. Because VLIW approaches make sense for wider processors, we choose to focus our example on such an architecture. For example, a VLIW processor might have instructions that contain five operations, including: one integer operation (which could also be a branch), two floating-point operations, and two memory references. The instruction would have a set of fields for each functional unit—perhaps 16 to 24 bits per unit, yielding an instruction length of between 112 and 168 bits.

To keep the functional units busy, there must be enough parallelism in a code sequence to fill the available operation slots. This parallelism is uncovered by unrolling loops and scheduling the code within the single larger loop body. If the unrolling generates straighline code, then *local scheduling* techniques, which operate on a single basic block can be used. If finding and exploiting the parallelism requires scheduling code across branches, a substantially more complex *global scheduling* algorithm must be used. Global scheduling algorithms are not only more complex in structure, but they must deal with significantly more complicated tradeoffs in optimization, since moving code across branches is expensive. T*race scheduling is* one of these global scheduling techniques

developed specifically for VLIWs.

Suppose we have a VLIW that could issue two memory references, two FP operations, and one integer operation or branch in every clock cycle. Show an unrolled version of the loop x[i] = x[i] +s (see page 223 for the MIPS ode) for such a processor. Unroll as many times as necessary to eliminate any stalls. Ignore the branch-delay slot.

The code is shown in Figure 4.5.The loop has been unrolled to make seven copies of the body, which eliminates all stalls (i.e., completely empty issue cycles), and runs in 9 cycles. This code yields a running rate of seven results in 9 cycles, or 1.29 cycles per result, nearly twice as fast as the two-issue superscalar that used unrolled and scheduled code.

Memory reference 1	**Memory reference 2**	**FP operation 1**	**FP operation 2**	**Integer operation/branch**
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D -8(R1),F8	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D -24(R1),F16			
S.D F20,-32(R1)	S.D -40(R1),F24			DADDUI R1,R1,#-56
S.D F28,8(R1)				BNE R1,R2,Loop

FIGURE 4.5 VLIW instructions that occupy the inner loop and replace the unrolled sequence.

For the original VLIW model, there are both technical and logistical problems. The technical problems are the increase in code size and the limitations of lock-step operation. Two different elements combine to increase code size substantially for a VLIW. First, generating enough operations in a straight-line code fragment requires ambitiously unrolling loops (as earlier examples) thereby increasing code size. Second, whenever instructions are not full, the unused functional units translate to wasted bits in the instruction encoding. In Figure 4.5, we saw that only about 60% of the functional units were used, so almost half of each instruction was empty. In most VLIWs, an instruction may need to be left completely empty if no operations can be scheduled.

Early VLIWs operated in lock-step; there was no hazard detection hardware at all. This structure dictated that a stall in any functional unit pipeline must cause the entire processor to stall, since all the functional units must be kept synchronized. Although a compiler may be able to schedule the deterministic functional units to prevent stalls, predicting which data accesses will encounter a cache stall and scheduling them is very difficult. Hence, caches needed to be blocking and to cause *all* the functional units to stall. As the issue rate and number of memory references becomes large, this synchronization restriction becomes unacceptable. In more recent processors, the functional units operate more independently, and the compiler is used to avoid hazards at issue time, while hardware checks allow for unsynchronized execution once instructions are issued.

Binary code compatibility has also been a major logistical problem for VLIWs. In a strict VLIW approach, the code sequence makes use of both the instruction set definition and the detailed pipeline structure, including both functional units and their latencies.
One possible solution to this migration problem, and the problem of binary code compatibility in general, is object-code translation or emulation. This technology is developing quickly and could play a significant role in future migration schemes. Another approach is to temper the strictness of the approach so that binary compatibility is still feasible. This later approach is used in the IA-64 architecture.

The major challenge for all multiple-issue processors is to try to exploit large amounts of ILP. When the parallelism comes from unrolling simple loops in FP programs, the original loop probably could have been run efficiently on a vector processor.

4Q. Explain how more parallelism can achieve at compiler time using Hardware

4.5 Hardware Support for Exposing More Parallelism at Compiler Time

Techniques such as loop unrolling, software pipelining, and trace scheduling can be used to increase the amount of parallelism available when the behavior of branches is fairly predictable at compile time. When the behavior of branches is not well known, compiler techniques alone may not be able to uncover much ILP. In such cases, the control dependences may severely limit the amount of parallelism that can be exploited. Similarly, potential dependences between memory reference instructions could prevent code movement that would increase available ILP. This section introduces several techniques that can help overcome such limitations.

The first is an extension of the instruction set to include *conditional* or *predicated instructions*. Such instructions can be used to eliminate branches converting a control dependence into a data dependence and potentially improving performance.

Hardware speculation with in-order commit preserved exception behavior by detecting and raising exceptions only at commit time when the instruction was no longer speculative. To enhance the ability of the *compiler* to speculatively move code over branches, while still preserving the exception behavior, we consider several different methods, which either include explicit checks for exceptions or techniques to ensure that only those exceptions that should arise are generated.

Finally, the hardware speculation schemes provided support for reordering loads and stores, by checking for potential address conflicts at runtime. To allow the compiler to reorder loads and stores when it suspects they do not conflict, but cannot be absolutely certain, a mechanism for checking for such conflicts can be added to the hardware. This mechanism permits additional opportunities for memory reference speculation.

Conditional or Predicated Instructions

The concept behind conditional instructions is quite simple: An instruction refers to a condition, which is evaluated as part of the instruction execution. If the condition is true,

the instruction is executed normally; if the condition is false, the execution continues as if the instruction was a no-op. The most common example of such an instruction is conditional move, which moves a value from one register to another if the condition is true. Such an instruction can be used to completely eliminate a branch in simple code sequences.

Consider the following code:

```
if (A==0) {S=T;}
```

Assuming that registers R1, R2, and R3 hold the values of A, S, and T, respectively,

The straightforward code using a branch for this statement is

```
    BNEZ R1,L
    ADDU R2,R3,R0
L:
```

Using a conditional move that performs the move only if the third operand is equal to zero, we can implement this statement in one instruction:

```
CMOVZ R2,R3,R1
```

The conditional instruction allows us to convert the control dependence present in the branch-based code sequence to a data dependence. For a pipelined processor, this moves the place where the dependence must be resolved from near the front of the pipeline, where it is resolved for branches, to the end of the pipeline where the register write occurs.

One obvious use for conditional move is to implement the absolute value function: A = abs (B), which is implemented as if (B<0) {A = - B;) else {A=B;}. This if statement can be implemented as a pair of conditional moves, or as one unconditional move (A=B) and one conditional move (A= - B).

In the example above or in the compilation of absolute value, conditional moves are used to change a control dependence into a data dependence. This enables us to eliminate

the branch and possibly improve the pipeline behavior.

Conditional moves are the simplest form of conditional or predicated instructions, and although useful for short sequences, have limitations. In particular, using conditional move to eliminate branches that guard the execution of large blocks of code can be in efficient, since many conditional moves may need to be introduced.

To remedy the in efficiency of using conditional moves, some architectures support full predication, whereby the execution of all instructions is controlled by a predicate. When the predicate is false, the instruction becomes a no-op. Full predication allows us to simply convert large blocks of code that are branch dependent. For example, an if-then-else statement within a loop can be entirely converted to predicated execution, so that the code in the then-case executes only if the value of the condition is true, and the code in the else-case executes only if the value of the condition is false. Predication is particularly valuable with global code scheduling, since it can eliminate nonloop branches, which significantly complicate instruction scheduling.

Predicated instructions can also be used to speculatively move an instruction that is time-critical, but may cause an exception if moved before a guarding branch. Although it is possible to do this with conditional move, it is more costly.

Predicated or conditional instructions are extremely useful for implementing short alternative control flows, for eliminating some unpredictable branches, and for reducing the overhead of global code scheduling. Nonetheless, the usefulness of conditional instructions is limited by several factors:

- Predicated instructions that are annulled (i.e., whose conditions are false) still take some processor resources. An annulled predicated instruction requires fetch resources at a minimum, and in most processors functional unit execution time.
- Predicated instructions are most useful when the predicate can be evaluated early. If the condition evaluation and predicated instructions cannot be separated (because of data dependences in determining the condition), then a conditional instruction may result in a stall for a data hazard. With branch prediction and speculation, such stalls can be avoided, at least when the branches are predicted accurately.
- The use of conditional instructions can be limited when the control flow involves

more than a simple alternative sequence. For example, moving an instruction across multiple branches requires making it conditional on both branches, which requires two conditions to be specified or requires additional instructions to compute the controlling predicate.

- Conditional instructions may have some speed penalty compared with unconditional instructions. This may show up as a higher cycle count for such instructions or a slower clock rate overall. If conditional instructions are more expensive, they will need to be used judiciously

.

For these reasons, many architectures have included a few simple conditional instructions (with conditional move being the most frequent), but only a few architectures include conditional versions for the majority of the instructions. The MIPS, Alpha, Power-PC, SPARC and Intel x86 (as defined in the Pentium processor) all support conditional move. The IA-64 architecture supports full predication for all instructions.

Compiler Speculation with Hardware Support

Many programs have branches that can be accurately predicted at compile time either from the program structure or by using a profile. In such cases, the compiler may want to speculate either to improve the scheduling or to increase the issue rate. Predicated instructions provide one method to speculate, but they are really more useful when control dependences can be completely eliminated by if-conversion. In many cases, we would like to move speculated instructions not only before branch, but before the condition evaluation, and predication cannot achieve this.

As pointed out earlier, to speculate ambitiously requires three capabilities:

1 the ability of the compiler to find instructions that, with the possible use of register renaming, can be speculatively moved and not affect the program data flow,

2 the ability to ignore exceptions in speculated instructions, until we know that such exceptions should really occur, and

3 the ability to speculatively interchange loads and stores, or stores and stores, which may have address conflicts.

The first of these is a compiler capability, while the last two require hardware support.

Hardware Support for Preserving Exception Behavior

There are four methods that have been investigated for supporting more ambitious speculation without introducing erroneous exception behavior:

1 The hardware and operating system cooperatively ignore exceptions for speculative instructions.

2 Speculative instructions that never raise exceptions are used, and checks are introduced to determine when an exception should occur.

3 A set of status bits, called *poison bits*, are attached to the result registers written by speculated instructions when the instructions cause exceptions. The poison bits cause a fault when a normal instruction attempts to use the register.

4 A mechanism is provided to indicate that an instruction is speculative and the hardware buffers the instruction result until it is certain that the instruction is no longer speculative.

To explain these schemes, we need to distinguish between exceptions that indicate a program error and would normally cause termination, such as a memory protection violation, and those that are handled and normally resumed, such as a page fault. Exceptions that can be resumed can be accepted and processed for speculative instructions just as if they were normal instructions. If the speculative instruction should not have been executed, handling the unneeded exception may have some negative performance effects, but it cannot cause incorrect execution. The cost of these exceptions may be high, however, and some processors use hardware support to avoid taking such exceptions, just as processors with hardware speculation may take some exceptions in speculative mode, while avoiding others until an instruction is known not to be speculative.

Exceptions that indicate a program error should not occur in correct programs, and the result of a program that gets such an exception is not well defined, except perhaps when the program is running in a debugging mode. If such exceptions arise in speculated

instructions, we cannot take the exception until we know that the instruction is no longer speculative.

In the simplest method for preserving exceptions, the hardware and the operating system simply handle all resumable exceptions when the exception occurs and simply return an undefined value for any exception that would cause termination.

A second approach to preserving exception behavior when speculating introduces speculative versions of instructions that do not generate terminating exceptions.

A third approach for preserving exception behavior tracks exceptions as they occur but postpones any terminating exception until a value is actually used, preserving the occurrence of the exception, although not in a completely precise fashion.

The fourth and final approach listed above relies on a hardware mechanism that operates like a reorder buffer. In such an approach, instructions are marked by the compiler as speculative and include an indicator of how many branches the instruction was speculatively moved across and what branch action (taken/not taken) the compiler assumed.

All instructions are placed in a reorder buffer when issued and are forced to commit in order, as in a hardware speculation approach. The reorder buffer tracks when instructions are ready to commit and delays the “write back” portion of any speculative instruction. Speculative instructions are not allowed to commit until the branches they have been speculatively moved over are also ready to commit, or, alternatively, until the corresponding sentinel is reached.

Hardware Support for Memory Reference Speculation

Moving loads across stores is usually done when the compiler is certain the addresses do not conflict. To allow the compiler to undertake such code motion, when it cannot be absolutely certain that such a movement is correct, a special instruction to check for address conflicts can be included in the architecture. The special instruction is left at the original location of the load instruction (and acts like a guardian) and the load is moved up across one or more stores.

When a speculated load is executed, the hardware saves the address of the accessed memory location. If a subsequent store changes the location before the check instruction,

then the speculation has failed. If the location has not been touched then the speculation is successful. Speculation failure can be handled in two ways. If only the load instruction was speculated, then it suffices to redo the load at the point of the check instruction If additional instructions that depended on the load were also speculated, then a fix-up sequence that re-executes all the speculated instructions starting with the load is needed.

5Q.Distinguish between Hardware versus Software Speculation Mechanisms

4.6 Crosscutting Issues

Hardware versus Software Speculation Mechanisms

Hardware Speculation	Software Speculation
Dynamic runtime disambiguation of memory addresses is done using Tomasulo's algorithm. This disambiguation allows us to move loads past stores at runtime.	Dynamic runtime disambiguation of memory addresses is difficult to do at compile time for integer programs that contain pointers
Hardware-based speculation works better when control flow is unpredictable, and when hardware-based branch prediction is superior to software-based branch prediction done at compile time.	Hardware-based branch prediction is superior than software-based branch prediction done at compile time.
Hardware-based speculation maintains a completely precise exception model even for speculated instructions	Software-based approaches have added special support to allow this as well.
Hardware-based speculation does not require compensation or bookkeeping code.	Software-based speculation require compensation or Bookkeeping
The ability to see further in the code is very poor in Hardware based speculation	Compiler-based approaches may benefit from the ability to see further in the code sequence, resulting in better code scheduling than a purely hardware-driven approach.

Hardware-based speculation with dynamic scheduling does not require different code sequences to achieve good performance	Software-based speculation with dynamic scheduling require different code sequences to achieve good performance
Speculation in hardware is complex and requires additional hardware resources	Speculation in Software is Simple