Instruction-Level Parallelism and Dynamic Exploitation

1Q. What is meant by Instruction Level Parallelism? Explain various techniques to implement ILP.

3.1 Instruction-Level Parallelism: Concepts and Challenges:

Instruction-level parallelism (ILP) is the potential overlap the execution of instructions using pipeline concept to improve performance of the system. The various techniques that are used to increase amount of parallelism are reduces the impact of data and control hazards and increases processor ability to exploit parallelism

There are two approaches to exploiting ILP.

1. Static Technique – Software Dependent
2. Dynamic Technique – Hardware Dependent

The static technique is compiler-intensive approach, which have broader adoption in the embedded market than the desktop or server markets, the new IA-64 architecture and Intel's Itanium, use this more static approach.

The dynamic technique is hardware intensive approach, which dominate the desktop and server markets and are used in a wide range of processors, including: the Pentium III and 4, the Althon, the MIPS R10000/12000, the Sun ultraSPARC III, the Power-PC 603, G3, and G4, and the Alpha 21264.

Technique	Reduces
Forwarding and bypassing	Potential data hazard stalls
Delayed branches and simple branch scheduling	Control hazard stalls
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences
Dynamic scheduling with renaming	Data hazard stalls and stalls from anti dependences and output dependences
Dynamic branch prediction	Control stalls
Issuing multiple instructions per cycle	Ideal CPI

Speculation	Data hazard and control hazard stalls
Dynamic memory disambiguation	Data hazard stalls with memory
Loop unrolling	Control hazard stalls
Basic compiler pipeline scheduling	Data hazard stalls
Compiler dependence analysis	Ideal CPI, data hazard stalls
Software pipelining, trace scheduling	Ideal CPI, data hazard stalls
Compiler speculation	Ideal CPI, data, control stalls

FIGURE 3.1 The major techniques together with the component of the CPI equation that the technique affects.

The simplest and most common way to increase the amount of parallelism is loop-level parallelism. Here is a simple example of a loop, which adds two 1000-element arrays, that is completely parallel:

```
for (i=1;i<=1000; i=i+1)
    x[i] = x[i] + y[i];
```

Every iteration of the loop can overlap with any other iteration, although within each loop iteration there is little or no opportunity for overlap.

There are a number of techniques for converting such loop-level parallelism into instruction-level parallelism are basically work by unrolling the loop either statically by the compiler or dynamically by the hardware. An important alternative method for exploiting loop-level parallelism is the use of vector instructions.

2Q. What is CPI? How to calculate Pipeline CPI.

CPI (Cycles per Instruction) for a pipelined processor is the sum of the base CPI and all contributions from stalls:

Pipeline CPI = Ideal pipeline CPI + Structural stalls + Data hazard stalls + Control stalls

The ideal pipeline CPI is a measure of the maximum performance attainable by the implementation. By reducing each of the terms of the right-hand side, we minimize the overall pipeline CPI and thus increase the IPC (Instructions per Clock).

3Q. Explain various types of Dependences in ILP.

Data Dependence and Hazards:

To exploit instruction-level parallelism, determine which instructions can be executed in parallel. If two instructions are parallel, they can execute simultaneously in a pipeline without causing any stalls. If two instructions are dependent they are not parallel and must be executed in order.

There are three different types of dependences: data dependences (also called true data dependences), name dependences, and control dependences.

Data Dependences:

An instruction j is data dependent on instruction i if either of the following holds:

- Instruction i produces a result that may be used by instruction j, or
- Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.

The second condition simply states that one instruction is dependent on another if there exists a chain of dependences of the first type between the two instructions. This dependence chain can be as long as the entire program.

For example, consider the following code sequence that increments a vector of values in memory (starting at 0(R1) and with the last element at 8(R2)) by a scalar in register F2:

```
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 (/e
       BNE R1,R2,LOOP        ; branch R1!=zero
```

The data dependences in this code sequence involve both floating point data:

```
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
```

and integer data:

```
DADDIU R1,R1,-8     ;decrement pointer
                    ;8 bytes (per DW)
BNE R1,R2,Loop      ; branch R1!=zero
```

Both of the above dependent sequences, as shown by the arrows, with each instruction depending on the previous one. The arrows here and in following examples show the order that must be preserved for correct execution. The arrow points from an instruction that must precede the instruction that the arrowhead points to.

If two instructions are data dependent they cannot execute simultaneously or be completely overlapped. The dependence implies that there would be a chain of one or more data hazards between the two instructions. Executing the instructions simultaneously will cause a processor with pipeline interlocks to detect a hazard and stall, thereby reducing or eliminating the overlap. Dependences are a property of programs. Whether a given dependence results in an actual hazard being detected and whether that hazard actually causes a stall are properties of the pipeline organization. This difference is critical to understanding how instruction-level parallelism can be exploited.

The presence of the dependence indicates the potential for a hazard, but the actual hazard and the length of any stall is a property of the pipeline. The importance of the data dependences is that a dependence (1) indicates the possibility of a hazard, (2) determines the order in which results must be calculated, and (3) sets an upper bound on how much parallelism can possibly be exploited. Such limits are explored in section 3.8.

Name Dependences

The name dependence occurs when two instructions use the same register or memory location, called a name, but there is no flow of data between the instructions associated with that name.

There are two types of name dependences between an instruction i that precedes instruction j in program order:

- An antidependence between instruction i and instruction j occurs when instruction j writes a register or memory location that instruction i reads. The original ordering must be preserved to ensure that i reads the correct value.
- An output dependence occurs when instruction i and instruction j write the same register or memory location. The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction j.

Both anti-dependences and output dependences are name dependences, as opposed to true data dependences, since there is no value being transmitted between the instructions. Since a name dependence is not a true dependence, instructions involved in a name dependence can execute simultaneously or be reordered, if the name (register number or memory location) used in the instructions is changed so the instructions do not conflict.

This renaming can be more easily done for register operands, where it is called register renaming. Register renaming can be done either statically by a compiler or dynamically by the hardware. Before describing dependences arising from branches, let's examine the relationship between dependences and pipeline data hazards.

Control Dependences:

A control dependence determines the ordering of an instruction, i, with respect to a branch instruction so that the instruction i is executed in correct program order. Every instruction, except for those in the first basic block of the program, is control dependent on some set of branches, and, in general, these control dependences must be preserved to preserve program order. One of the simplest examples of a control dependence is the dependence of the statements in the "then" part of an if statement on the branch. For example, in the code segment:

```
if  p1 {
        S1;
    };
if  p2 {
        S2;
    }
```

S1 is control dependent on p1, and S2is control dependent on p2 but not on p1. In general, there are two constraints imposed by control dependences:

1. An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch. For example, we cannot take an instruction from the then-portion of an if-statement and move it before the if-statement.
2. An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch. For example, we cannot take a statement before the if-statement and move it into the then-portion.

Control dependence is preserved by two properties in a simple pipeline, First, instructions execute in program order. This ordering ensures that an instruction that occurs before a branch is executed before the branch. Second, the detection of control or branch hazards ensures that an instruction that is control dependent on a branch is not executed until the branch direction is known.

4Q. What is Data Hazard? Explain various hazards in ILP.

Data Hazards

A hazard is created whenever there is a dependence between instructions, and they are close enough that the overlap caused by pipelining, or other reordering of instructions, would change the order of access to the operand involved in the dependence. Because of the dependence, preserve order that the instructions would execute in, if executed sequentially one at a time as determined by the original source program. The goal of both our software and hardware techniques is to exploit parallelism by preserving program order only where it affects the outcome of the program. Detecting and avoiding hazards ensures that necessary program order is preserved.

Data hazards may be classified as one of three types, depending on the order of read and write accesses in the instructions.

Consider two instructions i and j, with i occurring before j in program order. The possible data hazards are

RAW (read after write) — j tries to read a source before i writes it, so j incorrectly gets the old value. This hazard is the most common type and corresponds to a true data dependence. Program order must be preserved to ensure that j receives the value from i. In the simple common five-stage static pipeline a load instruction followed by an integer ALU instruction that directly uses the load result will lead to a RAW hazard.

WAW (write after write) — j tries to write an operand before it is written by i. The writes end up being performed in the wrong order, leaving the value written by i rather than the value written by j in the destination. This hazard corresponds to an output dependence. WAW hazards are present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled. The classic five-stage integer pipeline writes a register only in the WB stage and avoids this class of hazards.

WAR (write after read) — j tries to write a destination before it is read by i, so i incorrectly gets the new value. This hazard arises from an antidependence. WAR hazards cannot occur in most static issue pipelines even deeper pipelines or floating point pipelines because all reads are early (in ID) and all writes are late (in WB). A WAR hazard occurs either when there are some instructions that write results early in the instruction pipeline, and other instructions that read a source late in the pipeline or when instructions are reordered.

5Q. What is Dynamic Scheduling? Explain how it is used to reduce data hazards.

3.2 Overcoming Data Hazards with Dynamic Scheduling:

The Dynamic Scheduling is used handle some cases when dependences are unknown at a compile time. In which the hardware rearranges the instruction execution to reduce the stalls while maintaining data flow and exception behavior. it also allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline. Although a dynamically scheduled processor cannot change the data flow, it tries to avoid stalling when dependences, which could generate hazards, are present.

Dynamic Scheduling: The Idea

A major limitation of the simple pipelining techniques is that they all use in-order instruction issue and execution: Instructions are issued in program order and if an instruction is stalled in the pipeline, no later instructions can proceed. Thus, if there is a dependence between two closely spaced instructions in the pipeline, this will lead to a hazard and a stall. If there are multiple functional units, these units could lie idle. If instruction *j* depends on a long-running instruction *i,* currently in execution in the pipeline, then all instructions after *j* must be stalled until *i* is finished and *j* can execute. For example, consider this code:

```
DIV.D F0,F2,F4
ADD.D F10,F0,F8
SUB.D F12,F8,F14
```

The SUB.D instruction cannot execute because the dependence of ADD.D on DIV.D causes the pipeline to stall; yet SUB.D is not data dependent on anything in the pipeline. This hazard creates a performance limitation that can be eliminated by not requiring instructions to execute in program order.

In the classic five-stage pipeline both structural and data hazards could be checked during instruction decode (ID): When an instruction could execute without hazards, it was issued from ID knowing that all data hazards had been resolved. To allow us to begin executing the SUB.Din the above example, we must separate the issue process into two parts: checking for any structural hazards and waiting for the absence of a data hazard. We can still check for structural hazards when we issue the instruction; thus, we still use in-order instruction issue (i.e., instructions issue in program order), but we want an instruction to begin execution as soon as its data operand is available. Thus, this pipeline does *out-of-order execution*, which implies *out-of-order completion*.

Out-of-order execution introduces the possibility of WAR and WAW hazards, which do not exist in the five-stage integer pipeline and its logical extension to an in-order floating-point pipeline.

Out-of-order completion also creates major complications in handling exceptions. Dynamic scheduling with out-of-order completion must preserve exception behavior in

the sense that *exactly* those exceptions that would arise if the program were executed in strict program order *actually* do arise. Imprecise exceptions can occur because of two possibilities:

1. the pipeline may have already *completed* instructions that are *later* in program order than the instruction causing the exception, and
2. the pipeline may have *not yet completed* some instructions that are *earlier* in program order than the instruction causing the exception.

To allow out-of-order execution, we essentially split the ID pipe stage of our simple five-stage pipeline into two stages:

1 *Issue*—Decode instructions, check for structural hazards.

2 *Read operands*—Wait until no data hazards, then read operands.

In a dynamically scheduled pipeline, all instructions pass through the issue stage in order (in-order issue); however, they can be stalled or bypass each other in the second stage (read operands) and thus enter execution out of order. *Score-boarding* is a technique for allowing instructions to execute out-of-order when there are sufficient resources and no data dependences; it is named after the CDC 6600 scoreboard, which developed this capability. We focus on a more sophisticated technique, called *Tomasulo's algorithm*, that has several major enhancements over scoreboarding.

6Q. Explain Tomasulo's Approach how it is minimizes Data Hazards.

Dynamic Scheduling Using Tomasulo's Approach :

This scheme was invented by RobertTomasulo, and was first used in the IBM 360/91. it uses register renaming to eliminate output and anti-dependencies, i.e. WAW and WAR hazards. Output and anti-dependencies are just name dependencies, there is no actual data dependence. Tomasulo's algorithm implements register renaming through the use of what are called **reservation stations**. **Reservation stations** are buffers which fetch and store instruction operands as soon as they are available

In addition, pending instructions designate the reservation station that will provide their input. Finally, when successive writes to a register overlap in execution, only the last one is actually used to update the register. As instructions are issued, the register

specifies for pending operands are renamed to the names of the reservation station, which provides register renaming. Since there can be more reservation stations than real registers, the technique can even eliminate hazards arising from name dependences that could not be eliminated by a compiler.

The use of reservation stations, rather than a centralized register file, leads to two other important properties. First, hazard detection and execution control are distributed: The information held in the reservation stations at each functional unit determine when an instruction can begin execution at that unit. Second, results are passed directly to functional units from the reservation stations where they are buffered, rather than going through the registers. This bypassing is done with a common result bus that allows all units waiting for an operand to be loaded simultaneously (on the 360/91 this is called the *common data bus,* or CDB). In pipelines with multiple execution units and issuing multiple instructions per clock, more than one result bus will be needed.

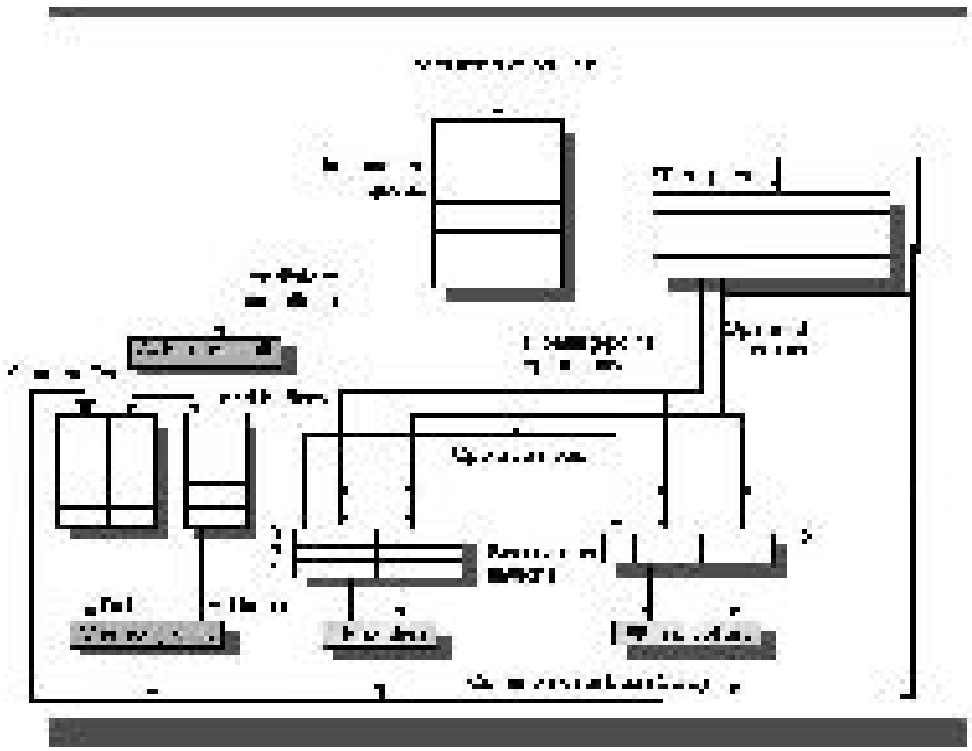


FIGURE 3.2 The basic structure of a MIPS floating point unit using Tomasulo's algorithm.

Figure 3.2 shows the basic structure of a Tomasulo-based MIPS processor, including both the floating-point unit and the load/store unit. Instructions are sent from the instruction unit into the instruction queue from which they are issued in FIFO order. The reservation stations include the operation and the actual operands, as well as information used for detecting and resolving hazards. Load buffers have three functions: hold the components of the effective address until it is computed, track outstanding loads that are waiting on the memory, and hold the results of completed loads that are waiting for the CDB. Similarly, store buffers have three functions: hold the components of the effective address until it is computed, hold the destination memory addresses of outstanding stores that are waiting for the data value to store, and hold the address and value to store until the memory unit is available. All results from either the FP units or the load unit are put on the CDB, which goes to the FP register file as well as to the reservation stations and store buffers. The FP adders implement addition and subtraction, and the FP multipliers do multiplication and division.

There are only three steps in Tomasulo's Aprroach :

1. *Issue*—Get the next instruction from the head of the instruction queue. If there is a matching reservation station that is empty, issue the instruction to the station with the operand values (renames registers)
2. *Execute(EX)*— When all the operands are available, place into the corresponding reservation stations for execution. If operands are not yet available, monitor the common data bus (CDB) while waiting for it to be computed.
3. *Write result (WB)*—When the result is available, write it on the CDB and from there into the registers and into any reservation stations (including store buffers) waiting for this result. Stores also write data to memory during this step: When both the address and data value are available, they are sent to the memory unit and the store completes.

Each reservation station has six fields:

- Op—The operation to perform on source operands S1 and S2.
- Qj, Qk—The reservation stations that will produce the corresponding source operand;

a value of zero indicates that the source operand is already available in Vj or Vk, or is unnecessary.

- Vj, Vk—The value of the source operands. Note that only one of the V field or the Q field is valid for each operand. For loads, the Vk field is used to the offset from the instruction.
- A–used to hold information for the memory address calculation for a load or store.
- Busy—Indicates that this reservation station and its accompanying functional unit are occupied.

7Q. How to Reduce Branch Costs with Dynamic Hardware Prediction

3.4. Reducing Branch Costs with Dynamic Hardware Prediction

Basic Branch Prediction and Branch-Prediction Buffers

The simplest dynamic branch-prediction scheme is a *branch-prediction buffer* or *branch history table*. A branch-prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not. if the prediction is correct—it may have been put there by another branch that has the same low-order address bits. The prediction is a hint that is assumed to be correct, and fetching begins in the predicted direction. If the hint turns out to be wrong, the prediction bit is inverted and stored back. The performance of the buffer depends on both how often the prediction is for the branch of interest and how accurate the prediction is when it matches.

This simple one-bit prediction scheme has a performance shortcoming: Even if a branch is almost always taken, we will likely predict incorrectly twice, rather than once, when it is not taken.

To remedy this, two-bit prediction schemes are often used. In a two-bit scheme, a prediction must miss twice before it is changed. Figure 3.7 shows the finite-state processor for a two-bit prediction scheme.

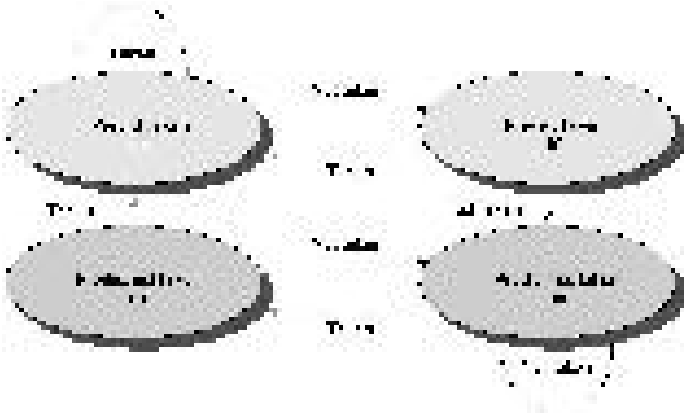


FIGURE 3.7 The states in a two-bit prediction scheme.

The two bits are used to encode the four states in the system. In a counter implementation, the counters are incremented when a branch is taken and decremented when it is not taken; the counters saturate at 00 or 11. One complication of the two-bit scheme is that it updates the prediction bits more often than a one-bit predictor, which only updates the prediction bit on a mispredict. Since we typically read the prediction bits on every cycle, a two-bit predictor will typically need both a read and a write access port.

The two-bit scheme is actually a specialization of a more general scheme that has an *n*-bit saturating counter for each entry in the prediction buffer. With an *n*-bit counter, the counter can take on values between 0 and $2^n - 1$: when the counter is greater than or equal to one half of its maximum value (2^{n-1}), the branch is predicted as taken; otherwise, it is predicted untaken. As in the two-bit scheme, the counter is incremented on a taken branch and decremented on an untaken branch. Studies of *n*-bit predictors have shown that the two-bit predictors do almost as well, and thus most systems rely on two-bit branch predictors rather than the more general *n*-bit predictors.

If the instruction is decoded as a branch and if the branch is predicted as taken, fetching begins from the target as soon as the PC is known. Otherwise, sequential fetching and executing continue. If the prediction turns out to be wrong, the prediction bits are changed as shown in Figure 3.7.

Although this scheme is useful for most pipelines, the five-stage, classic pipeline finds out both whether the branch is taken and what the target of the branch is at roughly the same time, *assuming* no hazard in accessing the register specified in the conditional branch.

To exploit more ILP, the accuracy of our branch prediction becomes critical, this problem in two ways: by increasing the size of the buffer and by increasing the accuracy of the scheme we use for each prediction.

Correlating Branch Predictors:

These two-bit predictor schemes use only the recent behavior of a single branch to predict the future behavior of that branch. It may be possible to improve the prediction accuracy if we also look at the recent behavior of *other* branches rather than just the branch we are trying to predict. Consider a small code fragment from the SPEC92 benchmark

```
if (aa==2)
aa=0;
if (bb==2)
bb=0;
if (aa!=bb) {
```

Here is the MIPS code that we would typically generate for this code fragment assuming that aa and bb are assigned to registers R1 and R2:

```
    DSUBUI R3,R1,#2
    BNEZ R3,L1 ;branch b1 (aa!=2)
    DADD R1,R0,R0 ;aa=0
L1: DSUBUI R3,R2,#2
    BNEZ R3,L2 ;branch b2(bb!=2)
    DADD R2,R0,R0 ; bb=0
L2: DSUBU R3,R1,R2 ;R3=aa-bb
    BEQZ R3,L3 ;branch b3 (aa==bb)
```

Let's label these branches b1, b2, and b3. The key observation is that the behavior of branch b3 is correlated with the behavior of branches b1 and b2. Clearly, if branches b1 and b2 are both not taken (i.e., the if conditions both evaluate to true and aa and bb are both assigned 0), then b3 will be taken, since aa and bb are clearly equal. A predictor that

uses only the behavior of a single branch to predict the outcome of that branch can never capture this behavior.

Branch predictors that use the behavior of other branches to make a prediction are called *correlating predictors* or *two-level predictors*.

Tournament Predictors: Adaptively Combining Local and Global Predictors

The primary motivation for correlating branch predictors came from the observation that the standard 2-bit predictor using only local information failed on some important branches and that by adding global information, the performance could be improved. Tournament predictors take this insight to the next level, by using multiple predictors, usually one based on global information and one based on local information, and combining them with a selector. Tournament predictors can achieve both better accuracy at medium sizes (8Kb-32Kb) and also make use of very large numbers of prediction bits effectively.

Tournament predictors are the most popular form of *multilevel branch predictors*. A multilevel branch predictor use several levels of branch prediction tables together with an algorithm for choosing among the multiple predictors; Existing tournament predictors use a 2-bit saturating counter per branch to choose among two different predictors. The four states of the counter dictate whether to use predictor 1 or predictor 2. The state transition diagram is shown in Figure 3.16.

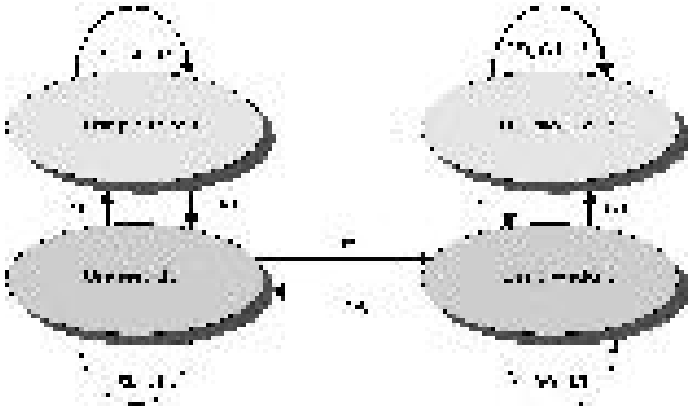
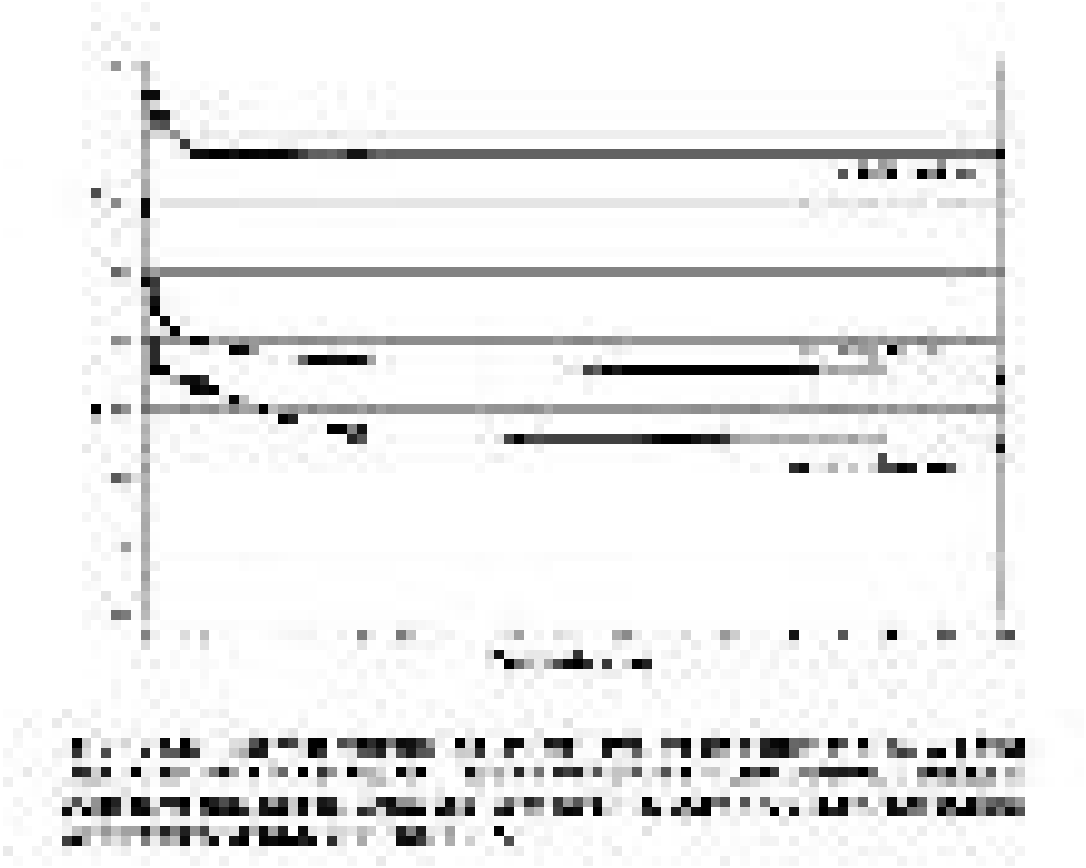


FIGURE 3.16 T**he state transition diagram for a tournament predictor has four states corresponding to which predictor to use.**

The advantage of a tournament predictor is its ability to select the right predictor for the right branch. Figure 3.17 shows how the tournament predictor selects between a local and global predictor depending on the benchmark, as well as on the branch. The ability to choose between a prediction based on strictly local information and one incorporating global information on a per branch basis is particularly critical in the integer benchmarks.

Figure 3.18 looks at the performance of three different predictors (a local 2-bit predictor, a correlating predictor, and a tournament predictor) for different numbers of bits using SPEC89 as the benchmark. As we saw earlier, the prediction capability of the local predictor does not improve beyond a certain size. The correlating predictor shows a significant improvement, and the tournament predictor generates slightly better performance.



8Q. Explain how to achieve high performance instruction delivery.

3.5. High Performance Instruction Delivery

Branch Target Buffers

A branch-prediction cache that stores the predicted address for the next instruction after a branch is called a branch-target buffer or branch-target cache.

For the classic, five-stage pipeline, a branch-prediction buffer is accessed during the ID cycle, so that at the end of ID we know the branch-target address (since it is computed during ID), the fall-through address (computed during IF), and the prediction. Thus, by the end of ID we know enough to fetch the next predicted instruction. For a branch-target buffer, we access the buffer during the IF stage using the instruction address of the fetched instruction, a possible branch, to index the buffer. If we get a hit, then we know the predicted instruction address at the end of the IF cycle, which is one cycle earlier than

for a branch-prediction buffer.

Because we are predicting the next instruction address and will send it out before decoding the instruction, we must know whether the fetched instruction is predicted as a taken branch. Figure 3.19 shows what the branch-target buffer looks like. If the PC of the fetched instruction matches a PC in the buffer, then the corresponding predicted PC is used as the next PC.

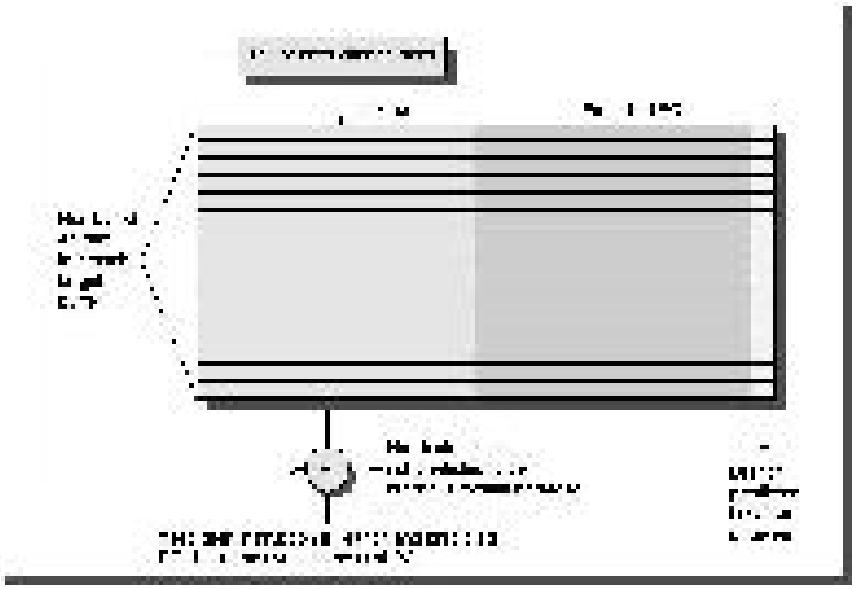


FIGURE 3.19 A branch-target buffer.

If a matching entry is found in the branch-target buffer, fetching begins immediately at the predicted PC. Note that the entry must be for this instruction, because the predicted PC will be sent out before it is known whether this instruction is even a branch. If we did not check whether the entry matched this PC, then the wrong PC would be sent out for instructions that were not branches, resulting in a slower processor. We only need to store the predicted-taken branches in the branch-target buffer, since an untaken branch follows the same strategy as a non branch.

Figure 3.20 shows the steps followed when using a branch-target buffer and where these steps occur in the pipeline. From this we can see that there will be no branch delay if a branch-prediction entry is found in the buffer and is correct.

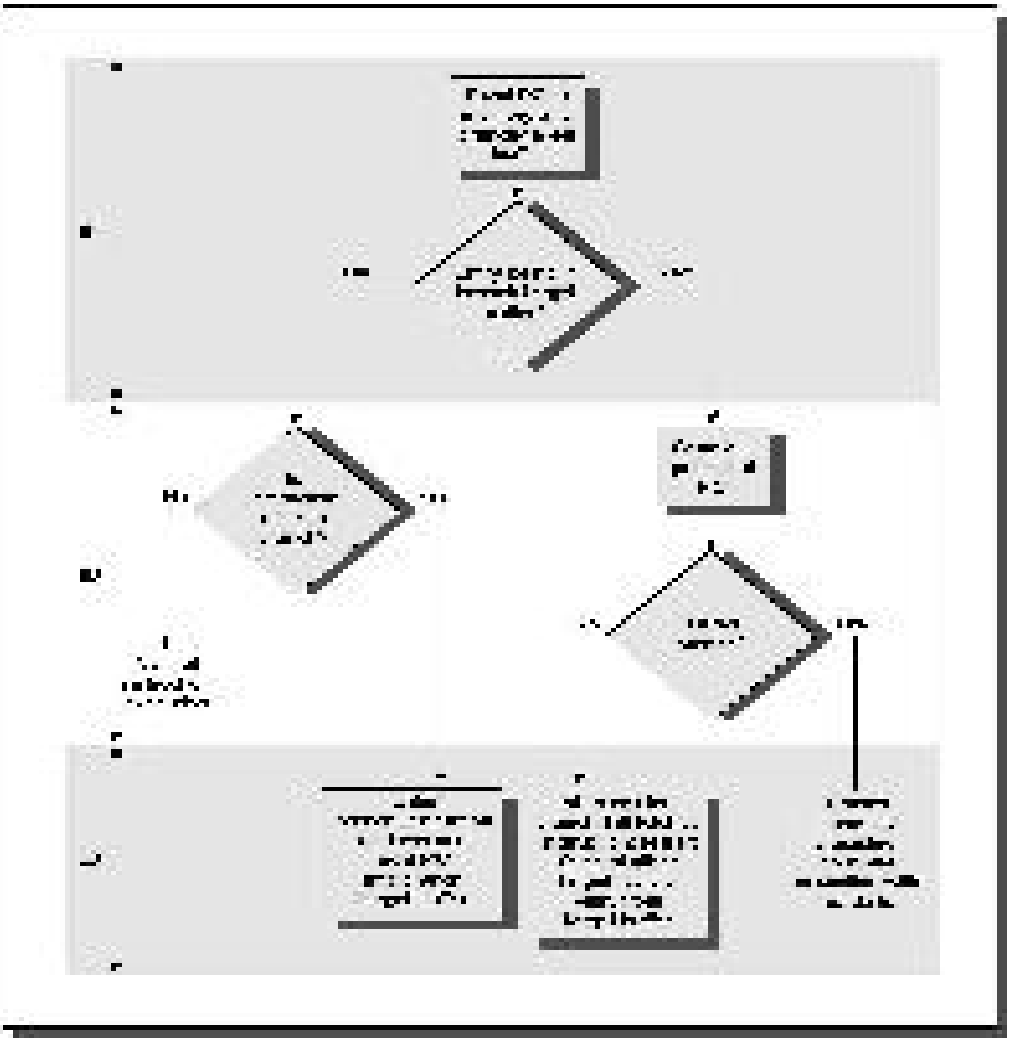


FIGURE 3.20 The steps involved in handling an instruction with a branch-target buffer

Integrated Instruction Fetch Units

The recent designs have used an integrated instruction fetch unit that integrates several functions:

1. Integrated branch prediction: the branch predictor becomes part of the instruction fetch unit and is constantly predicting branches, so to drive the fetch pipeline.
2. Instruction prefetch: to deliver multiple instructions per clock, the instruction fetch unit will likely need to fetch ahead. The unit autonomously manages the prefetching of instructions, integrating it with branch prediction.
3. Instruction memory access and buffering:.The instruction fetch unit also provides buffering, essentially acting as an on-demand unit to provide instructions to the issue stage as needed and in the quantity needed.

Return Address Predictors:
The concept of a small buffer of return addresses operating as a stack is used to predict the return address. This structure caches the most recent return addresses: pushing a return address on the stack at a call and popping one off at a return. If the cache is sufficiently large, it will predict the returns perfectly.

9Q. Explain the concept of Dynamic Prediction using Hardware based Speculation.

3.7. Hardware-Based Speculation

The type of code scheduling which executives instructions before or after the branch instruction which will not affect the program result is know n as Speculation. Hardware-based speculation combines three key ideas: dynamic branch prediction to choose which instructions to execute, speculation to allow the execution of instructions before the control dependences are resolved and dynamic scheduling to deal with the scheduling of different combinations of basic blocks. Hardware-based speculation follows the predicted flow of data values to choose when to execute instructions. This method of executing programs is essentially a *data-flow execution*: operations execute as soon as their operands are available.

The approach is implemented in a number of processors (PowerPC 603/604/G3/G4, MIPS R10000/R12000, Intel Pentium II/III/ 4, Alpha 21264, and AMD K5/K6/Athlon), is to implement speculative execution based on Tomasulo's algorithm.

The key idea behind implementing speculation is to allow instructions to execute out of order but to force them to commit *in order* and to prevent any irrevocable action until an instruction commits. In the simple single-issue five-stage pipeline we could ensure that instructions committed in order, and only after any exceptions for that instruction had been detected, simply by moving writes to the end of the pipeline. When we add speculation, we need to separate the process of completing execution from instruction commit, since instructions may finish execution considerably before they are ready to

commit. Adding this commit phase to the instruction execution sequence requires some changes to the sequence as well as an additional set of hardware buffers that hold the results of instructions that have finished execution but have not committed. This hardware buffer, which we call the *reorder buffer*, is also used to pass results among instructions that may be speculated.

The reorder buffer (ROB, for short) provides additional registers in the same way as the reservation stations in Tomasulo's algorithm extend the register set. The ROB holds the result of an instruction between the time the operation associated with the instruction completes and the time the instruction commits. Hence, the ROB is a source of operands for instructions, just as the reservation stations provide operands in Tomasulo's algorithm. The key difference is that in Tomasulo's algorithm, once an instruction writes its result, any subsequently issued instructions will find the result in the register file. With speculation, the register file is not updated until the instruction commits; thus, the ROB supplies operands in the interval between completion of instruction execution and instruction commit. The ROB is similar the store buffer in Tomasulo's algorithm, and we integrate the function of the store buffer into the ROB for simplicity.

Each entry in the ROB contains three fields: the instruction type, the destination field, and the value field. The instruction-type field indicates whether the instruction is a branch, a store, or a register operation. The destination field supplies the register number or the memory address, where the instruction result should be written. The value field is used to hold the value of the instruction result until the instruction commits.

Figure 3.29 shows the hardware structure of the processor including the ROB. The ROB completely replaces the store buffers.

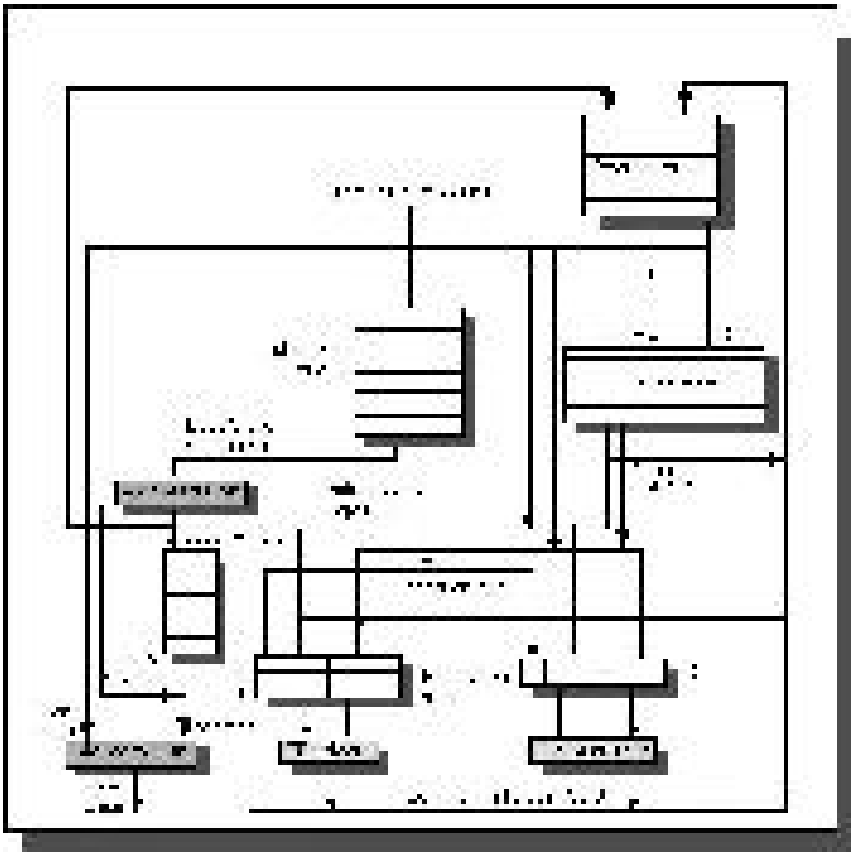


FIGURE 3.29 The basic structure of a MIPS FP unit using Tomasulo's algorithm and extended to handle speculation.

Here are the four steps involved in instruction execution:

1. *Issue*—Get an instruction from the instruction queue. Issue the instruction if there is an empty reservation station and an empty slot in the ROB, send the operands to the reservation station if they available in either the registers or the ROB for execution. If either all reservations are full or the ROB is full, then instruction issue is stalled until both have available entries. This stage is sometimes called *dispatch* in a dynamically scheduled processor.
2. *Execute*—If one or more of the operands is not yet available, monitor the CDB (common data bus) while waiting for the register to be computed. When both operands are available at a reservation station, execute the operation.
3. *Write result*—When the result is available, write it on the CDB and from the CDB into the ROB, as well as to any reservation stations waiting for this result. If the value to

be stored is not available yet, the CDB must be monitored until that value is broadcast, at which time the Value field of the ROB entry of the store is updated.

4. *Commit*—There are three different sequences of actions at commit depending on whether the committing instruction is: a branch with an incorrect prediction, a store, or any other instruction (normal commit). The normal commit case occurs when an instruction reaches the head of the ROB and its result is present in the buffer; at this point, the processor updates the register with the result and removes the instruction from the ROB. Committing a store is similar except that memory is updated rather than a result register. When a branch with incorrect prediction reaches the head of the ROB, it indicates that the speculation was wrong. The ROB is flushed and execution is restarted at the correct successor of the branch. If the branch was correctly predicted, the branch is finished. Some machines call this commit phase *completion* or *graduation.*

10Q. Explain various limitations of ILP.

3.8. Studies of the Limitations of ILP

The Hardware Model

An ideal processor is one where all artificial constraints on ILP are removed. The only limits on ILP in such a processor are those imposed by the actual data flows either through registers or memory.

The assumptions made for an ideal or perfect processor are as follows:

1. *Register renaming*—There are an infinite number of virtual registers available and hence all WAW and WAR hazards are avoided and an unbounded number of instructions can begin execution simultaneously.
2. *Branch prediction*—Branch prediction is perfect. All conditional branches are predicted exactly.
3. *Jump prediction*—All jumps (including jump register used for return and computed jumps) are perfectly predicted. When combined with perfect branch prediction, this is equivalent to having a processor with perfect speculation and an unbounded buffer of instructions available for execution.
4. *Memory-address alias analysis*—All memory addresses are known exactly and a load

can be moved before a store provided that the addresses are not identical.

Assumptions 2 and 3 eliminate *all* control dependences. Likewise, assumptions 1 and 4 eliminate *all but the true* data dependences. Together, these four assumptions mean that *any* instruction in the of the program's execution can be scheduled on the cycle immediately following the execution of the predecessor on which it depends.

Limitations on the Window Size and Maximum Issue Count

A dynamic processor might be able to more closely match the amount of parallelism uncovered by our ideal processor. consider what the perfect processor must do:

1. Look arbitrarily far ahead to find a set of instructions to issue, predicting all branches perfectly.
2. Rename all register uses to avoid WAR and WAW hazards.
3. Determine whether there are any data dependencies among the instructions in the issue packet; if so, rename accordingly.
4. Determine if any memory dependences exist among the issuing instructions and handle them appropriately.
5. Provide enough replicated functional units to allow all the ready instructions to issue.

Obviously, this analysis is quite complicated. For example, to determine whether *n* issuing instructions have any register dependences among them, assuming all instructions are register-register and the total number of registers is unbounded, requires

$$2n-2+2n-4+\ldots\ldots+2 = 2\sum_{i=1}^{n-1} i \quad = \quad [2\,(n-1)n]/2 \; = n^2 - n$$

comparisons. Thus, to detect dependences among the next 2000 instructions—the default size we assume in several figures—requires almost *four million* comparisons! Even issuing only 50 instructions requires 2450 comparisons. This cost obviously limits the number of instructions that can be considered for issue at once.

In existing and near-term processors, the costs are not quite so high, since we need only detect dependence pairs and the limited number of registers allows different solutions. Furthermore, in a real processor, issue occurs in-order and dependent instructions are handled by a renaming process that accommodates dependent renaming in one clock. Once instructions are issued, the detection of dependences is handled in a distributed fashion by the reservation stations or scoreboard.

The set of instructions that are examined for simultaneous execution is called the *window*. Each instruction in the window must be kept in the processor and the number of comparisons required every clock is equal to the maximum completion rate times the window size times the number of operands per instruction (today typically 6 x 80 x 2= 960), since every pending instruction must look at every completing instruction for either of its operands. Thus, the total window size is limited by the required storage, the comparisons, and a limited issue rate, which makes larger window less helpful. To date, the window size has been in the range of 32 to 126, which can require over 2,000 comparisons. The HP PA 8600 reportedly has over 7,000 comparators!

The window size directly limits the number of instructions that begin execution in a given cycle.

The Effects of Realistic Branch and Jump Prediction :

Our ideal processor assumes that branches can be perfectly predicted: The outcome of any branch in the program is known before the first instruction is executed. Figures 3.38 and 3.39 show the effects of more realistic prediction schemes in two different formats. Our data is for several different branch-prediction schemes varying from perfect to no predictor. We assume a separate predictor is used for jumps. Jump predictors are important primarily with the most accurate branch predictors, since the branch frequency is higher and the accuracy of the branch predictors dominates.

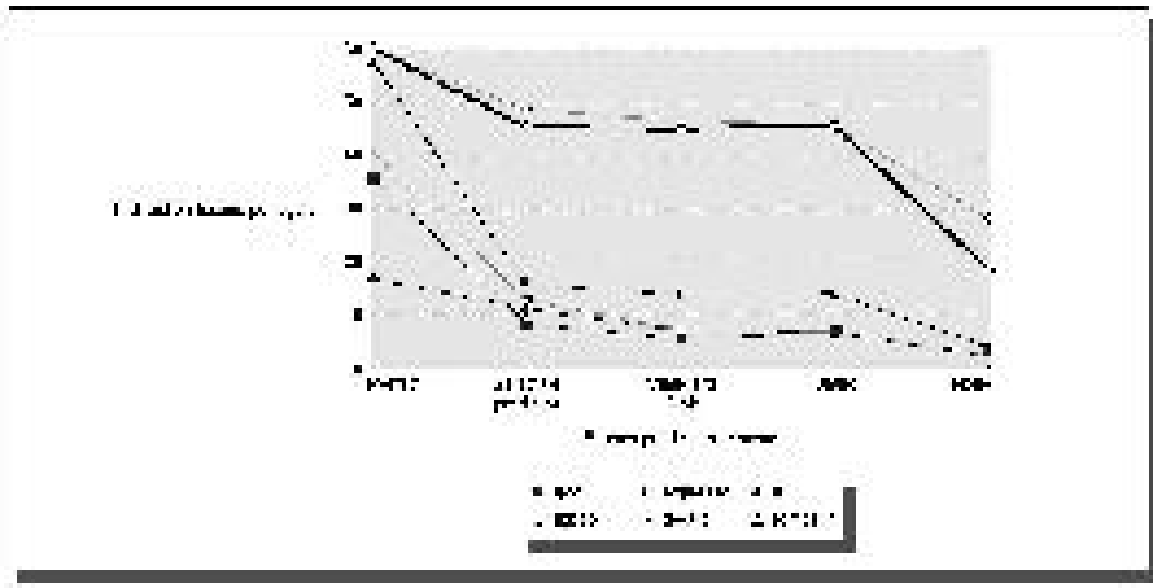
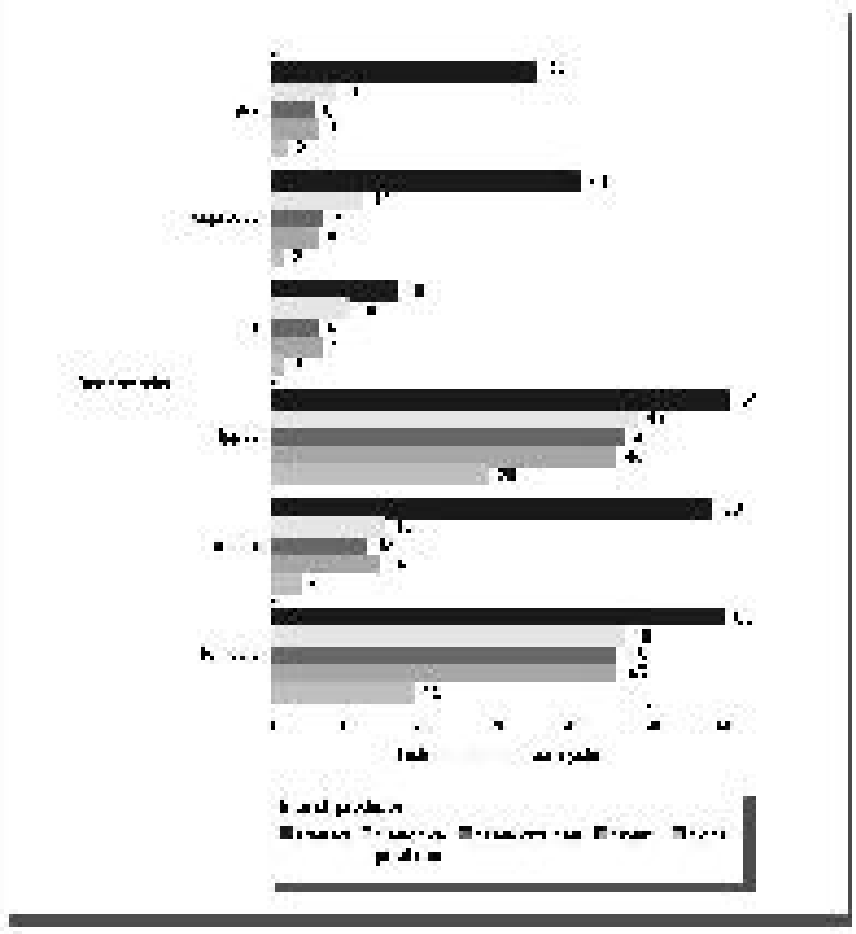


FIGURE 3.20 The distal internal carotid artery (ICA) is the most commonly affected site in ICA stenosis. Velocity increases as the stenosis becomes more severe. Peak systolic velocity is the most reliable criterion for grading ICA stenosis. Other criteria include end-diastolic velocity, the ratio of ICA velocity to common carotid artery velocity, and the presence or absence of poststenotic turbulence.



The five levels of branch prediction shown in these figures are

1. *Perfect*—All branches and jumps are perfectly predicted at the start of execution.
2. *Tournament-based branch predictor*—The prediction scheme uses a correlating two-bit predictor and a noncorrelating two-bit predictor together with a selector, which chooses the best predictor for each branch.
3. *Standard two-bit predictor with 512 two-bit entries*—In addition, we assume a 16-entry buffer to predict returns.
4. *Static*—A static predictor uses the profile history of the program and predicts that the branch is always taken or always not taken based on the profile.
5. *None*—No branch prediction is used, though jumps are still predicted. Parallelism is largely limited to within a basic block.

Branch prediction accuracy

The Effects of Finite Registers

Our ideal processor eliminates all name dependences among register references using an infinite set of physical registers. Figures 3.41 and 3.42 show the effect of reducing the number of registers available for renaming, again using the same data in two different forms. Both the FP and GP registers are increased by the number of registers shown on the axis or in the legend.

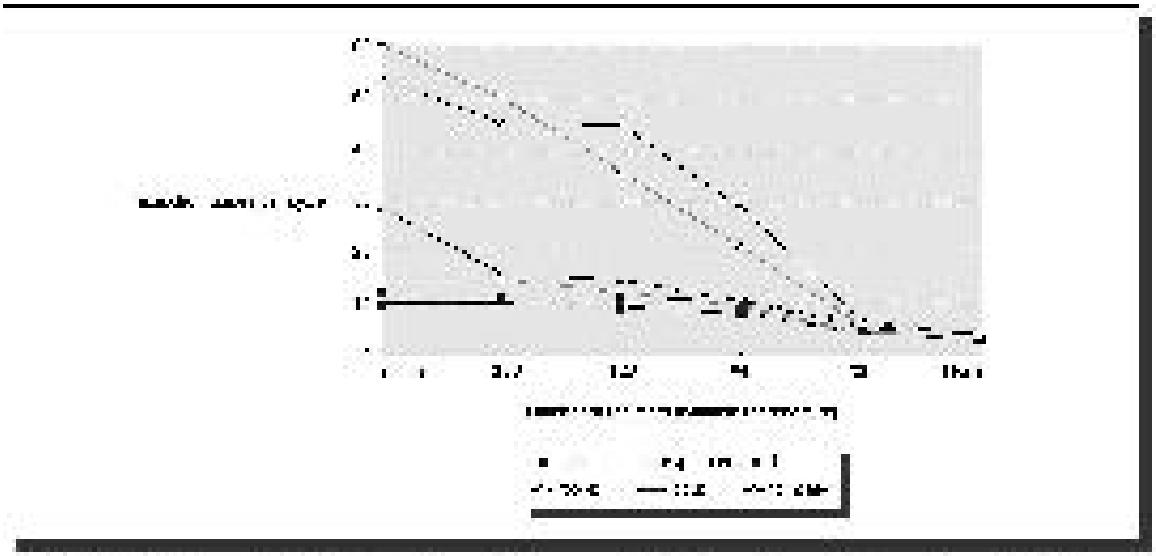


Figure 3.41 shows that the impact of having only a finite number of registers is significant if extensive parallelism exists. Although these graphs show a large impact on the floating-point programs, the impact on the integer programs is small primarily because the limitations in window size and branch prediction have limited the ILP substantially, making renaming less valuable. In addition, notice that the reduction in available parallelism is significant even if 64 additional integer and 64 additional FP registers are available for renaming, which is more than the number of extra registers available on any existing processor as of 2000.

Although register renaming is obviously critical to performance, an infinite number of registers is obviously not practical. Thus, for the next section, we assume that there are 256 integer and 256 FP registers available for renaming—far more than any anticipated

processor has.

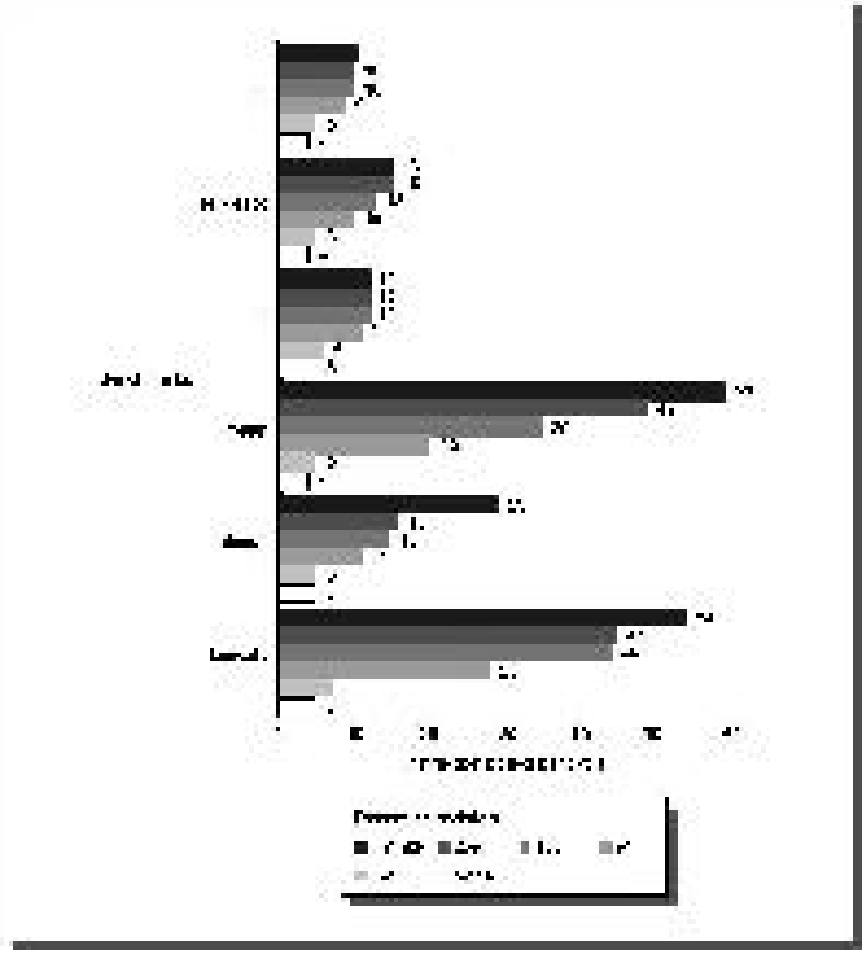
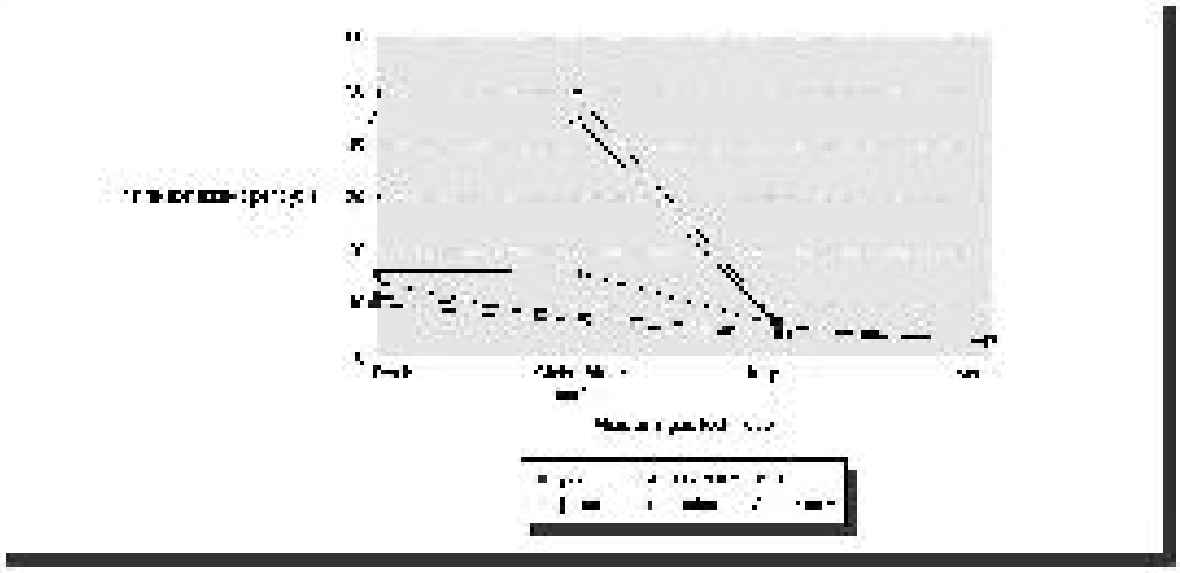
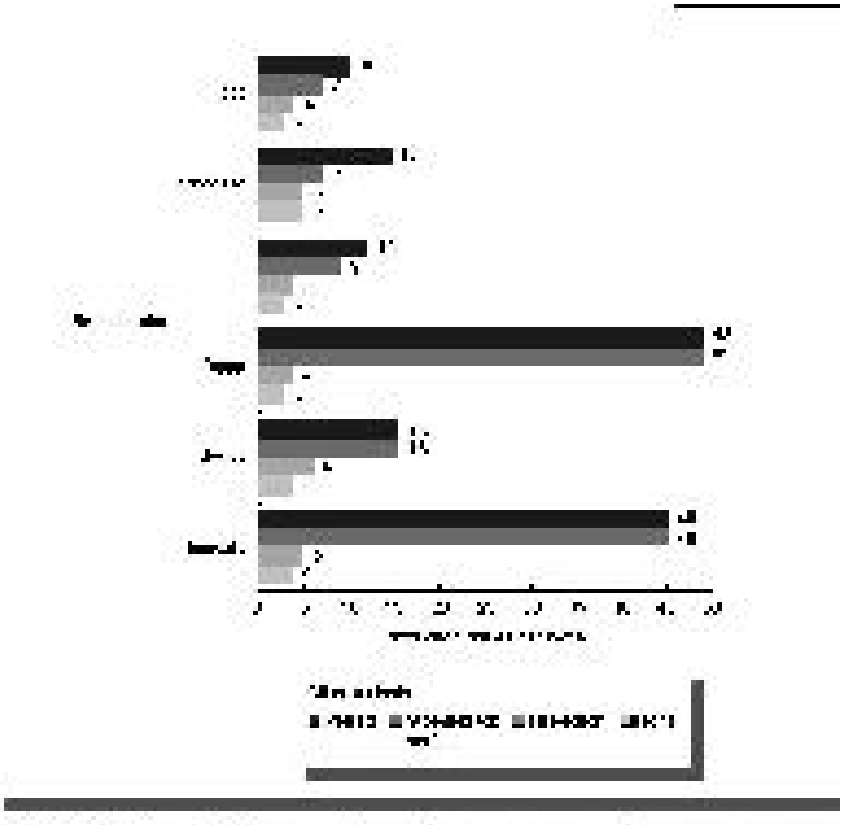


FIGURE 3.42 The reduction in available parallelism is significant when fewer than an unbounded number of renaming registers are available

The Effects of Imperfect Alias Analysis

Our optimal model assumes that it can perfectly analyze all memory dependences, as well as eliminate all register name dependences. Of course, perfect alias analysis is not possible in practice: The analysis cannot be perfect at compile time, and it requires a potentially unbounded number of comparisons at runtime





Figures 3.43 and 3.44 show the impact of three other models of memory alias analysis, in addition to perfect analysis. The three models are:

1 *Global/stack perfect*—This model does perfect predictions for global and stack references and assumes all heap references conflict. This model represents an idealized version of the best compiler-based analysis schemes currently in production. Recent and ongoing research on alias analysis for pointers should improve the handling of pointers to the heap in the future.

2 *Inspection*—This model examines the accesses to see if they can be determined not to interfere at compile time. For example, if an access uses R10 as a base register with an offset of 20, then another access that uses R10 as a base register with an offset of 100 cannot interfere. In addition, addresses based on registers that point to different allocation areas (such as the global area and the stack area) are assumed never to alias. This analysis is similar to that performed by many existing commercial compilers, though newer compilers can do better, at least for loop-oriented programs.

3. *None*—All memory references are assumed to conflict.

The dynamically scheduled processors rely on dynamic memory disambiguation and are limited by three factors:

1 To implement perfect dynamic disambiguation for a given load, we must know the memory addresses of all earlier stores that not yet committed, since a load may have a dependence through memory on a store. One technique for reducing this limitation on in-order address calculation is memory address speculation. With memory address speculation, the processor either assumes that no such memory dependences exist or uses a hardware prediction mechanism to predict if a dependence exists, stalling the load if a dependence is predicted. Of course, the processor can be wrong about the absence of the dependence, so we need a mechanism to discover if a dependence truly exists and to recover if so. To discover if a dependence exists, the processor examines the destination address of each completing store that is earlier in program order than the given load. If a dependence that should have been enforced occurs, the processor uses the speculative restart mechanism to redo the load and the following instructions. (We will see how this type of address speculation can be supported with instruction set extensions in the next chapter.)

2 Only a small number of memory references can be disambiguated per clock cycle.

3 The number of the load/store buffers determines how much earlier or later in the instruction stream a load or store may be moved.

Both the number of simultaneous disambiguations and the number of the load/ store buffers will affect the clock cycle time.

11Q. Write various limitation on ILP for Realizable Processors.

3.9 Limitations on ILP for Realizable Processors

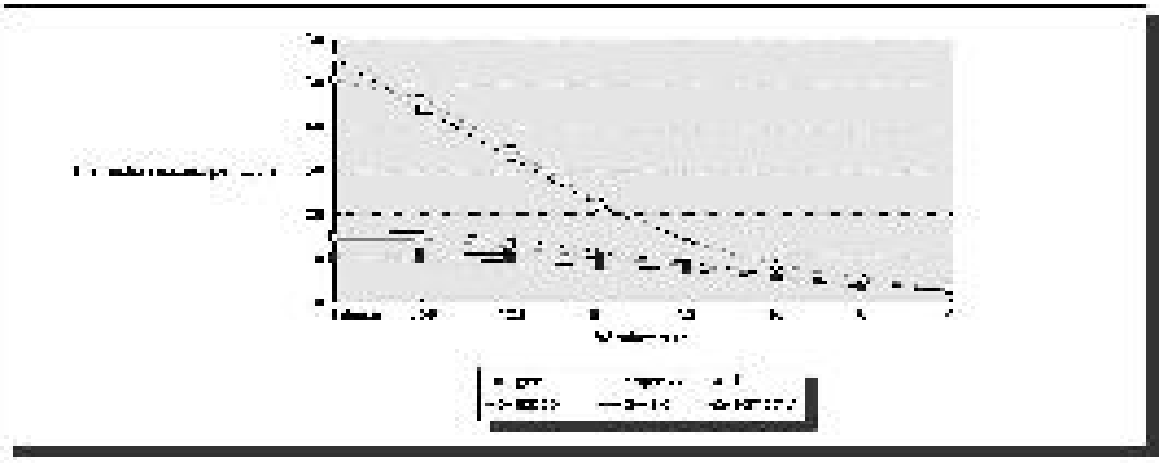
The performance of processors ambitious levels of hardware support equal to or better than what is likely in the next five years. In particular we assume the following fixed attributes:

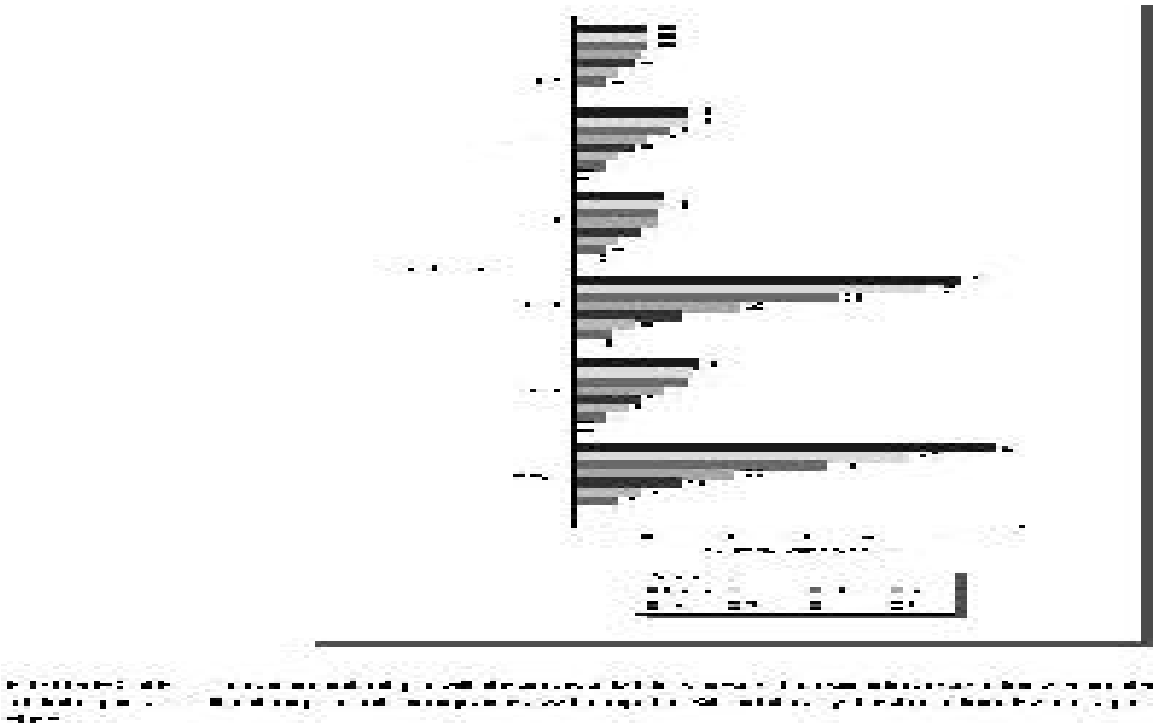
1. Up to 64 instruction issues per clock with *no* issue restrictions. As we discuss later, the practical implications of very wide issue widths on clock rate, logic complexity, and power may be the most important limitation on exploiting ILP.
2. A tournament predictor with 1K entries and a 16-entry return predictor. This predictor is fairly comparable to the best predictors in 2000; the predictor is not a primary bottleneck.
3. Perfect disambiguation of memory references done dynamically—this is ambitious but perhaps attainable for small window sizes (and hence small issue rates and load/store buffers) or through a memory dependence predictor.
4. Register renaming with 64 additional integer and 64 additional FP registers,exceeding largest number available on any processor in 2001 (41 and 41 in the Alpha 21264), but probably easily reachable within two or three years.

Figures 3.45 and 3.46 show the result for this configuration as we vary the window size.

Figure 3.45 shows the parallelism versus window size. The most startling observation is that with the realistic processor constraints listed above, the effect of the window size for the integer programs is not so severe as for FP programs. This result points to the key difference between these two types of programs. The availability of loop-level parallelism in two of the FP programs means that the amount of ILP that can be exploited

is higher, but that for integer programs other factors—such as branch prediction, register renaming, and less parallelism to start with—are all important limitations. This observation is critical, because of the increased emphasis on integer performance in the last few years. As we will see in the next section, for a realistic processor in 2000, the actual performance levels are much lower than those shown in Figure 3.45.





Beyond the limits of this study

limitations are two types that arise even for the perfect speculative processor and limitations that arise for one or more realistic models. Of course, all the limitations in the first class apply to the second. The most important limitations that apply even to the perfect model are:

1 *WAR and WAW hazards through memory:* the study eliminated WAW and WAR hazards through register renaming, but not in memory usage. Although, at first glance it might appear that such circumstances are rare (especially WAW hazards), they arise due to the allocation of stack frames. A called procedure reuses the memory locations of a previous procedure on the stack and this can lead to WAW and WAR hazards that are unnecessarily limiting. Austin and Sohi's 1992 paper examines this issue.

2 *Unnecessary dependences:* with infinite numbers of registers, all but true register data dependences are removed. There are, however, dependences arising from either recurrences or code generation conventions that introduce unnecessary true data dependences. One example of these is the dependence on the control variable in a simple do-loop: since the control variable is incremented on every loop iteration, the loop contains at least one dependence. As we show in the next chapter, loop unrolling and aggressive algebraic optimization can remove such dependent computation. Wall's study includes a limited amount of such optimizations, but applying them more aggressively could lead to increased amounts of ILP. In addition, certain code generation conventions introduce unneeded dependences, in particular the use of return address registers and a register for the stack pointer (which is incremented and decremented in the call/return sequence). Wall removes the effect of the return address register, but the use of a stack pointer in the linkage convention can cause "unnecessary" dependences. Postiff, Greene, Tyson, and Mudge explored the advantages of removing this constraint in a 1999 paper.

3 *Overcoming the data flow limit*: a recent proposed idea to boost ILP, which goes beyond the capability of the study above, is *value prediction*. Value prediction consists of predicting data values and speculating on the prediction. There are two obvious uses of this scheme: predicting data values and speculating on the result and predicting address values for memory alias elimination. The latter affects parallelism only under less than perfect circumstances,

4 Value prediction has possibly the most potential for increasing ILP. *Data value prediction and speculation* predicts data values and uses them in destination instructions speculatively. Such speculation allows multiple dependent instructions to be executed in the same clock cycle, thus increasing the potential ILP. To be effective, however, data values must be predicted very accurately, since they will be used by consuming instructions, just as if they were correctly computed. Thus, inaccurate prediction will lead to incorrect speculation and recovery, just as when branches are mispredicted.

One insight that gives some hope is that certain instructions produce the same values with high frequency, so it may be possible to selectively predict values for certain instructions with high accuracy. Obviously, perfect data value prediction would lead to infinite parallelism, since every value of every instruction could be predicted a priori.

Thus, studying the effect of value prediction in true limit studies is difficult and has not yet been done. Several studies have examined the role of value prediction in exploiting ILP in more realistic processors (e.g., Lipasti, Wilkerson, and Shen in 1996). The extent to which general value prediction will be used in real processors remains unclear at the present.

For a less than perfect processor, there are several ideas, which have been proposed, that could expose more ILP. We mention the two most important here:

1 *Address value prediction and speculation* predicts memory address values and speculates by reordering loads and stores. This technique eliminates the need to compute effective addresses in-order to determine whether memory references can be reordered, and could provide better aliasing analysis than any practical scheme. Because we need not actually predict data values, but only if effective addresses are identical, this type of prediction can be accomplished by simpler techniques. Recent processors include limited versions of this technique and it can be expected that future implementations of address value prediction may yield an approximation to perfect alias analysis, allowing processors to eliminate this limit to exploiting ILP.

2 Speculating on multiple paths: this idea was discussed by Lam and Wilson in 1992 and explored in the study covered in this section. By speculating on multiple paths, the cost of incorrect recovery is reduced and more parallelism can be uncovered. It only makes sense to evaluate this scheme for a limited number of branches, because the

hardware resources required grow exponentially. Wall's 1993 study provides data for speculating in both directions on up to eight branches. Whether such schemes ever become practical, or whether it will always be better to devote the equivalent silicon area to better branch predictors remains to be seen. In Chapter 8, we discuss thread-level parallelism and the use of speculative threads.

It is critical to understand that none of the limits in this section are fundamental in the sense that overcoming them requires a change in the laws of physics!
Instead, they are practical limitations that imply the existence of some formidable barriers to exploiting additional ILP. These limitations–whether they be window size, alias detection, or branch prediction–represent challenges for designers and researchers to overcome! As we discuss in the concluding remarks, there are a variety of other practical issues that may actually be the more serious limits to exploiting ILP in future processors.

www.jntuworld.com