INSTRUCTION LEVEL PARALLELISM

INTRODUCTION

Computer designers and computer architects have been striving to improve uniprocessor computer performance since the first computer was designed. The most significant advances in uniprocessor performance have come from exploiting advances in implementation technology. Architectural innovations have also played a part, and one of the most significant of these over The last decade has been the rediscovery of RISC architectures. Now that RISC architectures have gained acceptance both in scientific and marketing circles, computer architects have been thinking of new ways to improve uniprocessor performance. Many of these proposals such as VLIW, superscalar, and even relatively old ideas such as vector processing try to improve computer performance by exploiting instruction-level parallelism. They take advantage of this parallelism by issuing more than one instruction per cycle explicitly (as in VLIW or superscalar machines) or implicitly (as in vector machines). In this paper we will limit ourselves to improving uniprocessor performance, and will not discuss methods of improving application performance by using multiple processors in parallel. The amount of instruction-level parallelism varies widely depending on the type of code being executed. When we consider uniprocessor performance improvements due to exploitation of instruction-level parallelism, it is important to keep in mind the type of application environment. If the applications are dominated by highly parallel code (e.g., weather forecasting), any of a number of different parallel computers (e.g., vector, MIMD) would improve application performance. However, if the dominant applications have little instruction-level parallelism (e.g., compilers, editors, event-driven simulators, lisp interpreters), the performance improvements will be much smaller.

ILP HISTORY

Instruction-level Parallelism (ILP) is a critical technique used in computer architecture for processor and compiler design. ILP can improve the program execution performance by causing individual machine operations to execute in parallel. ILP appeared in the field of computer design 30 years ago. However it didn't play a important role in computer architecture design until 1980s. The quick development of the computer architecture technologies, ILP played a main role in designing a computer system, including the hardware and software design. So far, more CPU manufactures had incorporated ILP to their CPUs, and a bunch of new hardware and software techniques for ILP have become a popular topic. Actually when we talk about the ILP, it means how many instructions can be executed or issued at one time. Though we need hardware support for the ILP, the amount of availability of ILP is really got by the compiler before execution. So the available ILP in program became one of the central topics in computer compiler design in recent years. The study of how much instruction level parallelism actually exists in programs is a pretty interest field in processor architecture to boost the performance of a single processor by overlapping the execution of multiple instructions, using parallel processing models such as VLIW, superscalar, etc. This study attempts to measure the available parallelism in a program and tries to indicate whether the performance bottleneck is insufficient parallelism in the instruction stream. Its result will lead to reduce the instruction dependencies in the program by using appropriate compiler optimizations. So the study of ILP will do a great help to improve the performance of the nowadays' processor. And it can also make the program ILP

independent of the machine architecture. Program parallelism is very different from machine parallelism. If the program parallelism is low relative to machine parallelism, overall performance is limited by the program parallelism.

PARALLELISM

With the era of increasing processor speeds slowly coming to an end, computer architects are exploring new ways of increasing throughput. One of the most promising is to look for and exploit different types of parallelism in code.

TYPES OF PARALLELISM

There are three main types of parallelism.

Instruction Level Parallelism

Instruction level parallelism (ILP) takes advantage of sequences of instructions that require different functional units (such as the load unit, ALU, FP multiplier, etc). Different architectures approach this in different ways, but the idea is to have these non-dependent instructions executing simultaneously to keep the functional units busy as often as possible.

Data Level Parallelsim

Data level parallelism (DLP) is more of a special case than instruction level parallelism. DLP to the act of performing the same operation on multiple datum simultaneously. A classic example of DLP is performing an operation on an image in which processing each pixel is independent from the ones around it (such as brightening). This type of image processing lends itself well to having multiple pixels modified simultaneously using the same modification function. Other types of operations that allow the exploitation of DLP are matrix, array, and vector processing.

Thread Level Parallelism

Thread level parallelism (TLP) is the act of running multiple flows of execution of a single process simultaneously. TLP is most often found in applications that need to run independent, unrelated tasks (such as computing, memory accesses, and IO) simultaneously. These types of applications are often found on machines that have a high workload, such as web servers. TLP is a popular ground for current research due to the rising popularity of multi-core and multi-processor systems, which allow for different threads to truly execute in parallel.

INSTRUCTION LEVEL PARALLELISM

DEFINITION

Abbreviated as ILP, ***I**nstruction-**L**evel **P**arallelism* is a measurement of the number of operations that can be performed simultaneously in a computer program. Microprocessors exploit ILP by executing multiple instructions from a single program in a single cycle.

EXPLANTION

Instruction-level parallelism (ILP) is a measure of how many of the operations in a computer program can be performed simultaneously. Consider the following program:

1. e = a + b
2. f = c + d
3. g = e * f

Operation 3 depends on the results of operations 1 and 2, so it cannot be calculated until both of them are completed. However, operations 1 and 2 do not depend on any other operation, so they can be calculated simultaneously. If we assume that each operation can be completed in one unit of time then these three instructions can be completed in a total of two units of time, giving an ILP of 3/2.

A goal of compiler and processor designers is to identify and take advantage of as much ILP as possible. Ordinary programs are typically written under a sequential execution model where instructions execute one after the other and in the order specified by the programmer. ILP allows the compiler and the processor to overlap the execution of multiple instructions or even to change the order in which instructions are executed.

How much ILP exists in programs is very application specific. In certain fields, such as graphics and scientific computing the amount can be very large. However, workloads such as cryptography exhibit much less parallelism.

DATA DEPENDANCY

A **data dependency** in computer science is a situation in which a program statement (instruction) refers to the data of a preceding statement.

Dependences are a property of programs. If two instructions are data dependent they cannot execute simultaneously. A dependence results in a hazard and the hazard causes a stall. Data dependences may occur through registers or memory

TYPES OF DEPENDENCY

1.Name dependence

Two instructions use the same register/memory (name), but there is no flow

of data.

There are further two types of name dependency

1) Anti-dependence
2) Output dependence

Output dependence

An output dependency occurs when the ordering of instructions will affect the final output value of a variable. In the example below, there is an output dependency between instructions 3 and 1; changing the ordering of instructions in this example will change the final value of B, thus these instructions cannot be executed in parallel.

1 A = 2 * X
2 B = A / 3
3 A = 9 * Y

Anti-dependence

An anti-dependency occurs when an instruction requires a value that is later updated. In the following example, instruction 3 anti-depends on instruction 2the ordering of these instructions cannot be changed, nor can they be executed in parallel (possibly changing the instruction ordering), as this would affect the final value of A.

1. B = 3
2. A = B + 1
3. B = 7

2.Data True dependence

A true dependency, also known as a data dependency, occurs when an instruction depends on the result of a previous instruction:

1. A = 3
2. B = A
3. C = B

Instruction 3 is truly dependent on instruction 2, as the final value of C depends on the instruction updating B. Instruction 2 is truly dependent on instruction 1, as the final value of B depends on the instruction updating A. Since instruction 3 is truly dependent upon instruction 2 and instruction 2 is truly dependent on instruction 1, instruction 3 is also truly dependent on instruction 1. Instruction level parallelism is therefore not an option in this example.

3.Control Dependence

Any instruction B is control dependent on a preceding instruction A if the latter determines whether B should execute or not. In the following example, instruction 2 is control dependent on instruction 1.

1. If a == b goto AFTER

2. A = 2 * X4.

3. AFTER:

4.Resource Dependence

An instruction is resource-dependent on a previously issued instruction if it requires a hardware resource which is still being used by a previously issued instruction e.g.

div r1, r2, r3

div r4, r2, r5

TYPES OF HAZARDS

1.Data hazards

A hazard is created whenever there is dependence between instructions, and they are close enough that the overlap caused by pipelining would change the order of access to an operand. Or we can say that data hazards occur when instructions which exhibit data dependence modify data in different stages of a pipeline. Data hazards make the performance lower. The situation when the next instruction depends on the results of the previous one is occurred very often. It means that these instructions cannot be executed together. There are three situations in which a data hazard can occur:

1. Read after write (RAW), a true dependency
2. Write after read (WAR)
3. Write after write (WAW)

Read After Write (RAW)

A RAW Data Hazard refers to a situation where we refer to a result that has not yet been calculated or retrieved

RAW data hazard is the most common type. It arises when the next instruction tries to read a source before the previous instruction writes to it. So, the next instruction gets the old value incorrectly

For example:

1. R2 <- R1 + R3
2. R4 <- R2 + R3

The first instruction is calculating a value to be saved in register 2, and the second is going to use this value to compute a result for register 4. However, in a pipeline, when we fetch the operands for the 2nd operation, the results from the first will not yet have been saved, and hence we have a data dependency.

We say that there is a data dependency with instruction 2, as it is dependent on the completion of instruction 1.

Write After Read (WAR)

A WAR Data Hazard represents a problem with concurrent execution.

WAR hazard arises when the next instruction writes to a destination before the previous instruction reads it. In this case, the previous instruction gets a new value incorrectly

For example:

1. R4 <- R1 + R3
2. R3 <- R1 + R2

If we are in a situation that there is a chance that 2 may be completed before 1 (i.e. with concurrent execution) we must ensure that we do not store the result of register 3 before 1 has had a chance to fetch the operands.

Write After Write (WAW)

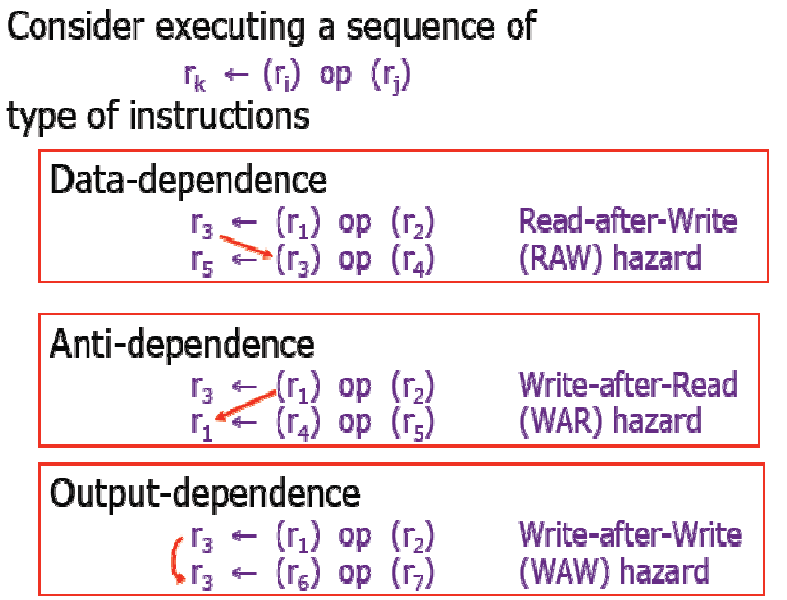
A WAW Data Hazard is another situation which may occur in a concurrent execution environment.

For example:

1. R2 <- R1 + R2
2. R2 <- R4 + R7

We must delay the WB (Write Back) of 2 until the execution of 1.

The overall picture of the data hazard is described in a picture below



2.Structural hazards

A structural hazard occurs when a part of the processor's hardware is needed by two or more instructions at the same time. A canonical example is a single memory unit that is accessed both in the fetch stage where an instruction is retrieved from memory, and the memory stage where data is written and/or read from memory.

3.Control hazards (branch hazards)

Branching hazards (also known as control hazards) occur with branches. On many instruction pipeline microarchitectures, the processor will not know the outcome of the branch when it needs to insert a new instruction into the pipeline.

HARDWARE SUPPORT FOR ILP

We can see there are several mechanisms for supporting the ILP by hardware: first, using multiple, parallel functional units and second, pipelining the functional units . And also we can use dynamic scheduling, such as scoreboarding or Tomasulo approach, to reduce the

dependencies among program. Superscalar and VLIW are the basic hardware techniques for exploiting the ILP. Superscalar and VLI processors can potentially provide large performance improvements over their scalar predecessors by providing multiple data paths and functional units. The parallel resourses are exploited by concurrently executing independent instructions from the instruction stream. However, conditional branch instructions pose difficult problems for all types of processors that exploit ILP. Recent studies have shown that by using conventional code optimization and scheduling methods, superscalar and VLIW processors cannot produce a sustained speedup of more than two for nonnumeric programs. For such programs, conventional architectural and compilation methods do not provide enough support to utilize these processors.

COMPILER SUPPORT FOR ILP

As we know, the ILP is exploited both by compiler and hardware support. However compiler provides the
inherent and implicit ILP in program to hardware by compilation optimization. There is lots of compiler
techniques for extracting the available ILP in programs:

- Scheduling
- register allocation and renaming.
- Loop unroll
- control-flow analysis and optimization:Branch prediction and speculation.
- memory access optimization: value prediction.

TECHNIQUES FOR ILP

TECHNIQUES	REDUCES
Forwarding and bypassing	Potential data hazard stall
Delayed branches & branch scheduling	Control hazard stalls
Basic dynamic scheduling (scoreboarding)	Data hazards from true dependencies
Dynamic scheduling with renaming	Data hazards from antideps and output deps.
Dynamic branch prediction	Control stalls
Speculation	Data and control hazard stalls
Dynamic memory disambiguation	Data hazard stalls with memory
Dynamic memory disambiguation	Data hazard stalls with memory