Advanced Topics in Computer Architecture

Lecture 4
Limits on Instruction Level Parallelism

Marenglen Biba
Department of Computer Science
University of New York Tirana

Outline

- Introduction
- Studies of the Limitations of ILP
- Limitations on ILP for Realizable Processors
- Crosscutting Issues: Hardware versus Software Speculation
- Multithreading: Using ILP Support to Exploit Thread-Level Parallelism
- Putting It All Together: Performance and Efficiency in Advanced Multiple-Issue Processors
- Fallacies and Pitfalls

Introduction

- Exploiting ILP was the primary focus of processor designs for about 20 years starting in the mid-1980s.
- For the first 15 years, we saw a progression of successively more sophisticated schemes for pipelining, multiple issue, dynamic scheduling and speculation.
- Since 2000, designers have focused primarily on optimizing designs or trying to achieve higher clock rates without increasing issue rates.
- As we indicated in the previous lecture, this era of advances in exploiting ILP appears to be coming to an end.

Introduction

- We will examine the limitations on ILP from program structure, from realistic assumptions about hardware budgets, and from the accuracy of important techniques for speculation such as branch prediction.
- We will examine the use of thread-level parallelism as an alternative or addition to instruction-level parallelism.
- Finally, we conclude the lecture by comparing a set of recent processors both in performance and in efficiency measures per transistor and per watt.

Outline

- Introduction
- Studies of the Limitations of ILP
- Limitations on ILP for Realizable Processors
- Crosscutting Issues: Hardware versus Software Speculation
- Multithreading: Using ILP Support to Exploit Thread-Level Parallelism
- Putting It All Together: Performance and Efficiency in Advanced Multiple-Issue Processors
- Fallacies and Pitfalls

Studies of the Limitations of ILP

- Exploiting ILP to increase performance began with the first pipelined processors in the 1960s.
- In the 1980s and 1990s, these techniques were key to achieving rapid performance improvements.
- The question of how much ILP exists was critical to our long-term ability to enhance performance at a rate that exceeds the increase in speed of the base integrated circuit technology.
- On a shorter scale, the critical question of what is needed to exploit more ILP is crucial to both computer designers and compiler writers.

Studies and Assumptions

- Several studies of available parallelism operate by *making a set of assumptions* and seeing how much parallelism is available under those assumptions.
- The data we will examine here come from a study that makes the *fewest assumptions*; in fact, the ultimate hardware model is probably unrealizable.
- Nonetheless, all such studies assume a certain level of compiler technology, and some of these assumptions could affect the results, despite the use of incredibly ambitious hardware.

Studies and Assumptions

- In the future, advances in compiler technology together with significantly new and different hardware techniques may be able to overcome some limitations assumed in these studies;

  However, it is unlikely that such advances when coupled with realistic hardware will overcome these limits in the near future.
- For example, *value prediction*, which we examined in the last chapter, can remove data dependence limits.
  - For value prediction to have a significant impact on performance, however, predictors would need to achieve far higher prediction accuracy than has so far been reported.
- Indeed for reasons we will discuss here we are likely reaching the limits of how much ILP can be exploited efficiently.

The Hardware Model

- To see what the limits of ILP might be, we first need to define an ideal processor.
- An ideal processor is one where all constraints on ILP are removed. The only limits on ILP in such a processor are those imposed by the actual data flows through either 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.

The Hardware Model

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. Note that this implements perfect address alias analysis.

5. *Perfect caches*

   — All memory accesses take 1 clock cycle. In practice, superscalar processors will typically consume large amounts of ILP hiding cache misses, making these results highly optimistic.

The Hardware Model

- 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 program's execution can be scheduled *on the cycle immediately following* the execution of the predecessor on which it depends.
- It is even possible, under these assumptions, for the *last* dynamically executed instruction in the program to be scheduled on the very first cycle!
- Thus, this set of assumptions subsumes both control and address speculation and implements them as if they were perfect.

The Hardware Model

- Initially, we examine a processor that can issue an unlimited number of instructions at once looking arbitrarily far ahead in the computation.
- For all the processor models we examine, there are no restrictions on what types of instructions can execute in a cycle.
- For the unlimited-issue case, this means there may be an unlimited number of loads or stores issuing in 1 clock cycle. In addition, all functional unit latencies are assumed to be 1 cycle, so that any sequence of dependent instructions can issue on successive cycles.
- Latencies longer than 1 cycle would decrease the number of issues per cycle, although not the number of instructions under execution at any point.

The Hardware Model and IBM Power5

- The instructions in execution at any point are often referred to as *in flight*.
- *The ideal processor is on the edge of unrealizable.*
- For example, the IBM Power5 is one of the most advanced superscalar processors announced to date.
- The Power5 issues up to four instructions per clock and initiates execution on up to six (with significant restrictions on the instruction type, e.g., at most two load-stores), supports a large set of renaming registers (88 integer and 88 floating point, allowing over 200 instructions in flight, of which up to 32 can be loads and 32 can be stores), uses a large aggressive branch predictor, and employs dynamic memory disambiguation.
- After looking at the parallelism available for the perfect processor, we will examine the impact of restricting various features.

Measuring Performance

- To measure the available parallelism, a set of programs was compiled and optimized with the standard MIPS optimizing compilers.

  The programs were instrumented and executed to produce a trace of the instruction and data references.
- Every instruction in the trace is then scheduled as early as possible, limited only by the data dependences.
- Since a trace is used, perfect branch prediction and perfect alias analysis are easy to do.
- With these mechanisms, instructions may be scheduled much earlier than they would otherwise, moving across large numbers of instructions on which they are not data dependent, including branches, since branches are perfectly predicted.

ILP available in a perfect processor

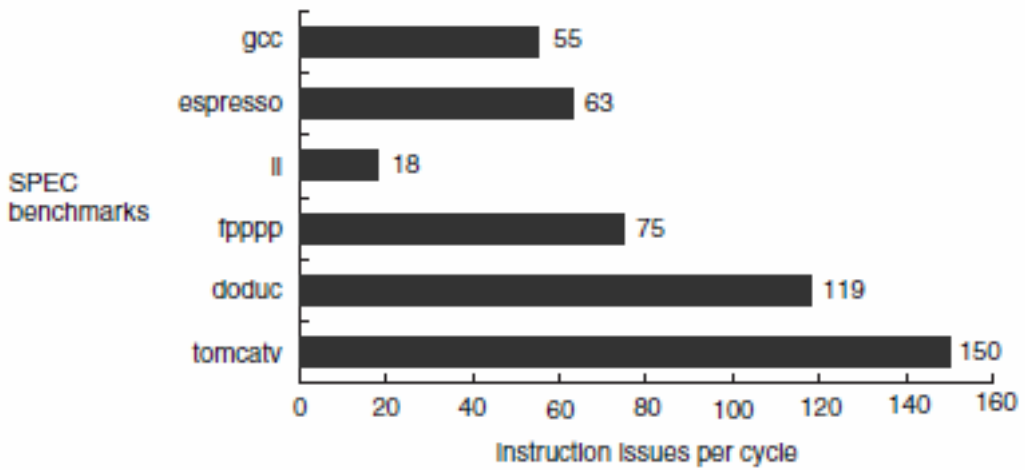

Figure 3.1 ILP available in a perfect processor for six of the SPEC92 benchmarks. The first three programs are integer programs, and the last three are floating-point programs. The floating-point programs are loop-intensive and have large amounts of loop-level parallelism.

Throughout this lecture the parallelism is measured by the average instruction issue rate.

Next, we will restrict various aspects of this processor to show what the effects of various assumptions are before looking at some ambitious but realizable processors.

Limitations on the Window Size and Maximum Issue Count

- To build a processor that even comes close to perfect branch prediction and perfect alias analysis requires extensive dynamic analysis, since static compile time schemes cannot be perfect.
- Of course, most realistic dynamic schemes will not be perfect, but the use of dynamic schemes will provide the ability to uncover parallelism that cannot be analyzed by static compile time analysis.

  Thus, a dynamic processor might be able to more closely match the amount of parallelism uncovered by our ideal processor.

Approaching the ideal processor

- How close could a real dynamically scheduled, speculative processor come to the ideal processor? To gain insight into this question, 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 dependences 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.

Complexity of Analysis

- Obviously, the dynamic 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 these number of comparisons:

$$2n-2+2n-4+\ldots+2=2\sum_{i=1}^{n-1} i=2\frac{(n-1)n}{2}=n^2-n$$

- Thus, to detect dependences among the next 2000 instructions — the default size we assume in several figures — requires almost *4 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.

Complexity in real processors

- 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 1 clock.
- Once instructions are issued, the detection of dependences is handled in a distributed fashion by the reservation stations or scoreboard.

Window

- The set of instructions that is 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

  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 a larger window less helpful.

Window Size

- The window size directly limits the number of instructions that begin execution in a given cycle.
- In practice, real processors will have a more limited number of functional units (e.g., no superscalar processor has handled more than two memory references per clock), as well as limited numbers of buses and register access ports, which serve as limits on the number of instructions initiated per clock.
- Thus, the maximum number of instructions that may issue, begin execution, or commit in the same clock cycle is usually much smaller than the window size.

Implementation constraints

- Obviously, the number of possible implementation constraints in a multiple-issue processor is large, including:
  - issues per clock
  - functional units and unit latency
  - register file ports
  - functional unit queues (which may be fewer than units),
  - issue limits for branches
  - limitations on instruction commit.
- Each of these acts as a constraint on the ILP.
- Rather than try to understand each of these effects, however, we will focus on limiting the size of the window, with the understanding that all other restrictions would further reduce the amount of parallelism that can be exploited.

Restricting the size of the window

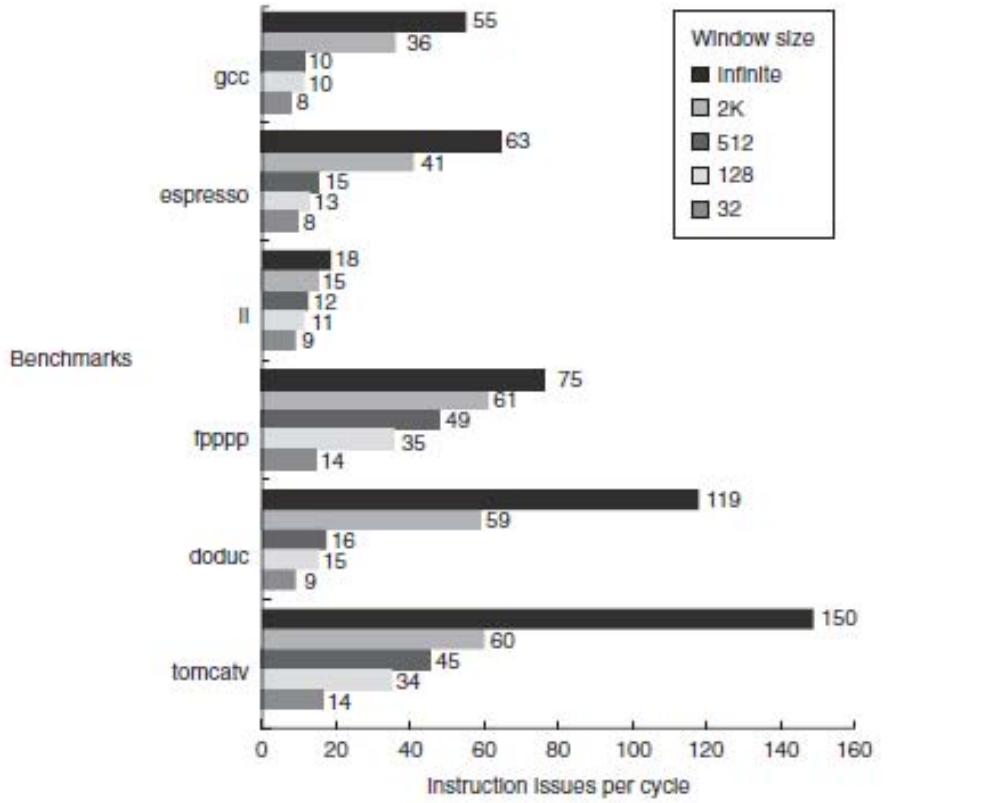

Figure 3.2 The effect of window size shown by each application by plotting the average number of instruction issues per clock cycle.

As we can see the amount of parallelism uncovered falls sharply with decreasing window size.

Restricting the size of the window

- In 2005, the most advanced processors have window sizes in the range of 64–200, but these window sizes are not strictly comparable to those shown in the previous Figure for two reasons.
  - First, many functional units have multicycle latency, reducing the effective window size compared to the case where all units have single-cycle latency.
  - Second, in real processors the window must also hold any memory references waiting on a cache miss, which are not considered in this model, since it assumes a perfect, single-cycle cache access.

Real windows

- We know that very large window sizes are impractical and inefficient, and the data in Figure 3.2 tells us that instruction throughput will be *considerably reduced* with realistic implementations.
- Thus, we will assume a base window size of 2K entries, roughly 10 times as large as the largest implementation in 2005, and a maximum issue capability of 64 instructions per clock, also 10 times the widest issue processor in 2005, for the rest of this analysis.
- As we will next, when the rest of the processor is not perfect, a 2K window and a 64-issue limitation do not constrain the amount of ILP the processor can exploit.

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!*
- Of course, no real processor can ever achieve this.
- Analysis
  - Our data are 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.

The Effects of Realistic Branch and Jump Prediction

Predictors in analysis:

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 2-bit predictor and a noncorrelating 2-bit predictor together with a selector, which chooses the best predictor for each branch.
3. *Standard 2-bit predictor with 512 2-bit entries*. In addition, we assume a 16-entry buffer to predict returns.
4. *Profile-based* —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.

Effect of branch-prediction schemes

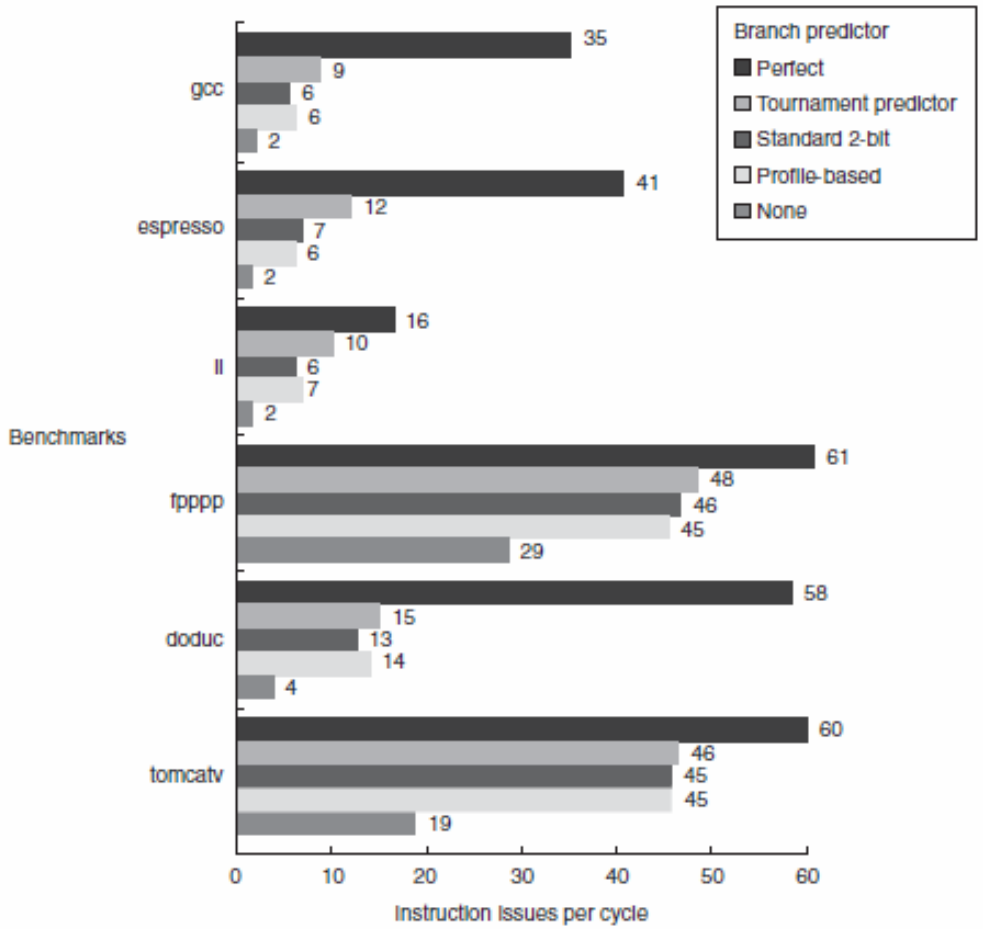

Figure 3.3 The effect of branch-prediction schemes sorted by application. This graph shows the impact of going from a perfect model of branch prediction (all branches predicted correctly arbitrarily far ahead); to various dynamic predictors (selective and 2-bit); to compile time, profile-based prediction; and finally to using no predictor. The predictors are described precisely in the text. This graph highlights the differences among the programs with extensive loop-level parallelism (tomcatv and fpppp) and those without (the integer programs and doduc).

Realistic predictors for conditional branches

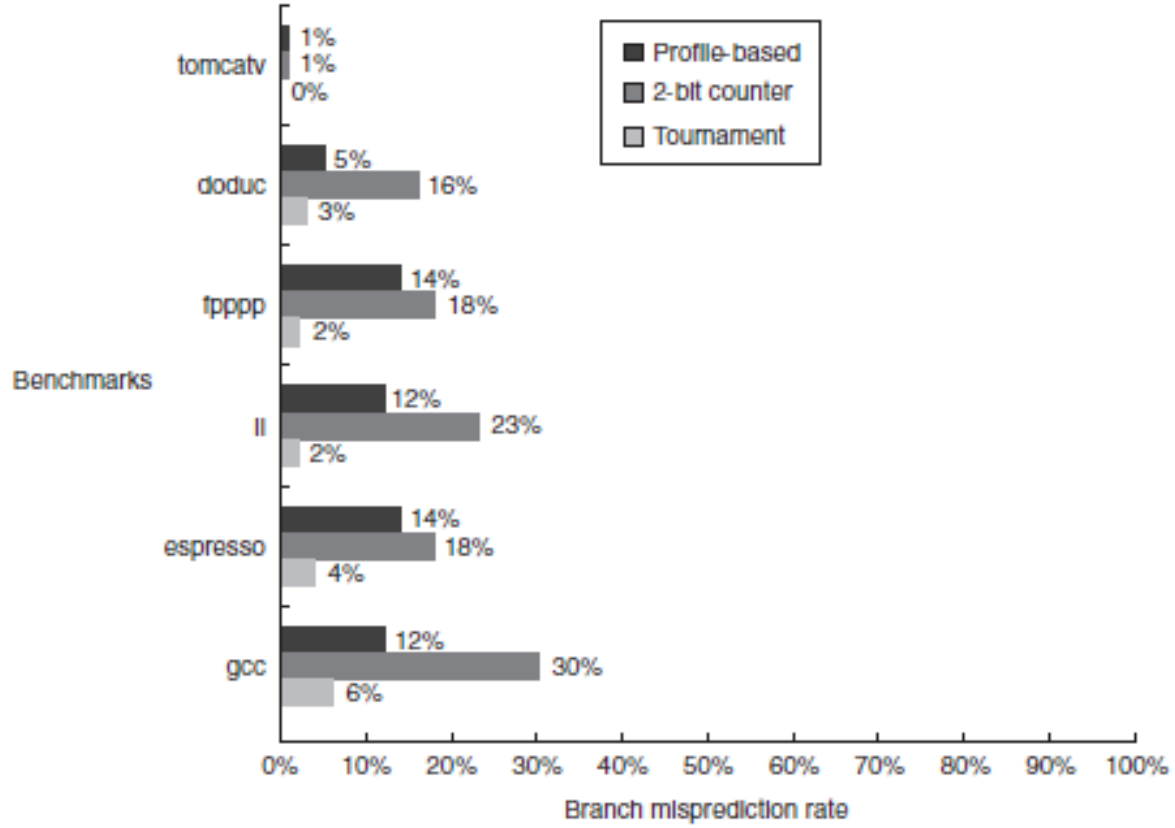

Figure 3.4 Branch misprediction rate for the conditional branches in the SPEC92 subset.

Refining Analysis

- As we have seen, branch prediction is critical, especially with a window size of 2K instructions and an issue limit of 64.
- For the rest of this section, in addition to the window and issue limit, we assume as a base a more ambitious tournament predictor that uses two levels of prediction and a total of 8K entries.
- This predictor, which requires more than 150K bits of storage (roughly four times the largest predictor to date), slightly outperforms the selective predictor described above (by about 0.5 – 1%).
- We also assume a pair of 2K jump and return predictors, as described above.

The Effects of Finite Registers

- Our ideal processor eliminates all name dependences among register references using an infinite set of virtual registers.
- To date, the IBM Power5 has provided the largest numbers of virtual registers: 88 additional floating-point and 88 additional integer registers, in addition to the 64 registers available in the base architecture.
- All 240 registers are shared by two threads when executing in multithreading mode, and all are available to a single thread when in single-thread mode.
- Figure in next slide shows the effect of reducing the number of registers available for renaming; *both* the FP and GP registers are increased by the number of registers shown in the legend.

Reducing the number of registers

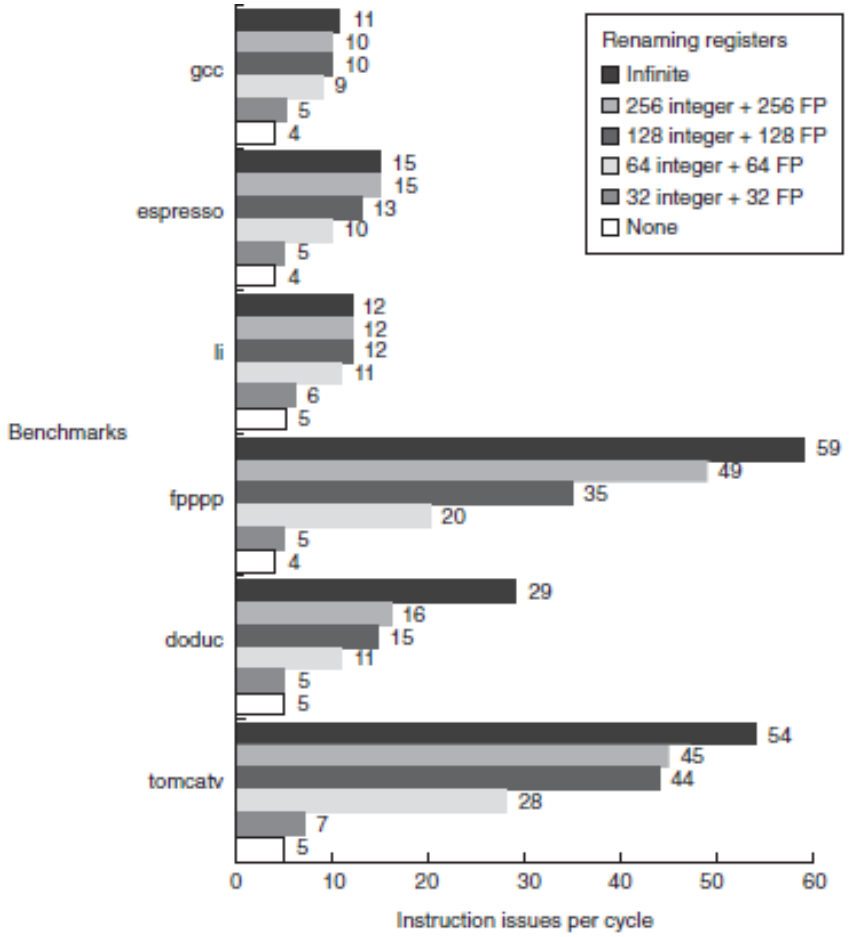

Figure 3.5 The reduction in available parallelism is significant when fewer than an unbounded number of renaming registers are available. *Both* the number of FP registers and the number of GP registers are increased by the number shown on the *x*-axis. So, the entry corresponding to "128 integer + 128 FP" has a total of 128 + 128 + 64 = 320 registers (128 for integer renaming, 128 for FP renaming, and 64 integer and FP registers present in the MIPS architecture). The effect is most dramatic on the FP programs, although having only 32 extra integer and 32 extra FP registers has a significant impact on all the programs. For the integer programs, the impact of having more than 64 extra registers is not seen here. To use more than 64 registers requires uncovering lots of parallelism, which for the integer programs requires essentially perfect branch prediction.

Choosing the number of registers

- Although register renaming is obviously critical to performance, an infinite number of registers is not practical.
- Thus, for the next slides, we assume that there are 256 integer and 256 FP registers available for renaming — far more than any anticipated processor has as of 2005.

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 run time (since the number of simultaneous memory references is unconstrained).
- There are three models of memory alias analysis, in addition to perfect analysis. The three models are:

Models of memory alias analysis

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.
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, assuming R10 could not have changed.
3. *None*—All memory references are assumed to conflict.

The effect of varying levels of alias analysis

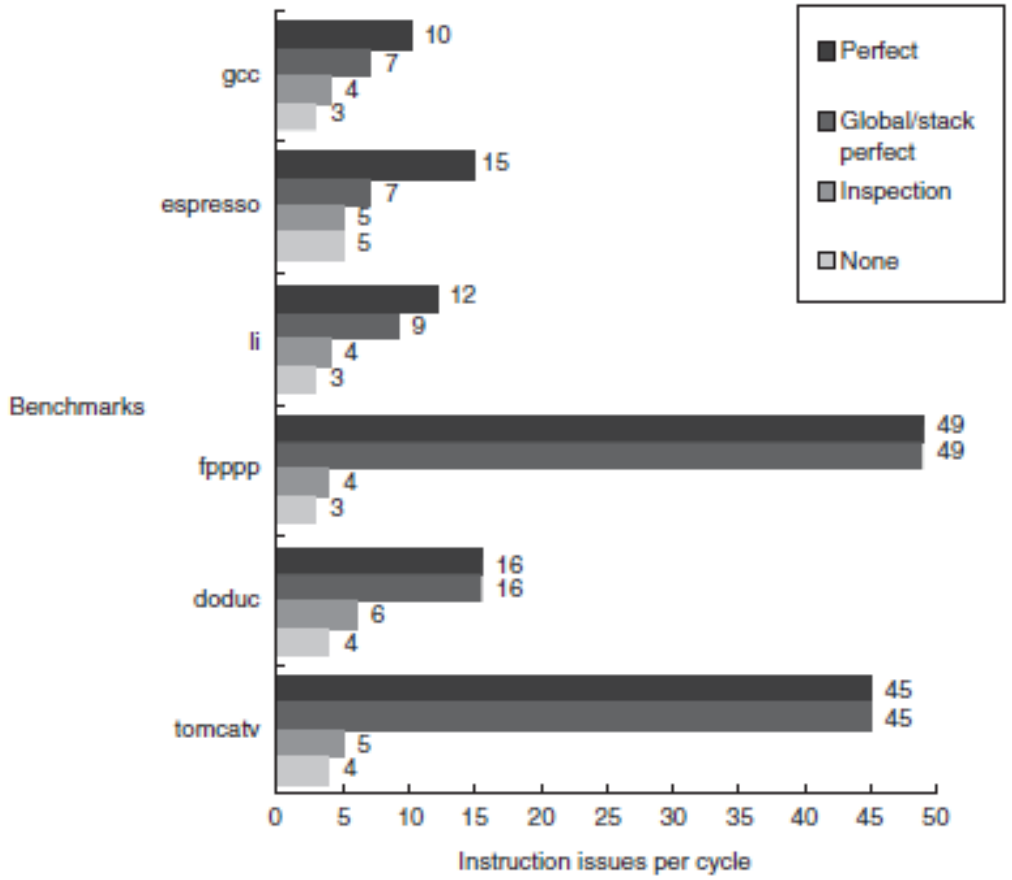

Figure 3.6 The effect of varying levels of alias analysis on individual programs. Anything less than perfect analysis has a dramatic impact on the amount of parallelism found in the integer programs, and global/stack analysis is perfect (and unrealizable) for the FORTRAN programs.

Outline

- Introduction
- Studies of the Limitations of ILP
- Limitations on ILP for Realizable Processors
- Crosscutting Issues: Hardware versus Software Speculation
- Multithreading: Using ILP Support to Exploit Thread-Level Parallelism
- Putting It All Together: Performance and Efficiency in Advanced Multiple-Issue Processors
- Fallacies and Pitfalls

Hardware Support

- Here we look at the performance of processors with ambitious levels of hardware support equal to or better than what is available in 2006 or likely to be available in the next few years.
- In particular we assume the following fixed attributes:

1. Up to 64 instruction issues per clock with *no* issue restrictions, or roughly 10 times the total issue width of the widest processor in 2005. 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 2005; the predictor is not a primary bottleneck.

Hardware Support

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, which is roughly comparable to the IBM Power5.

Varying the window size

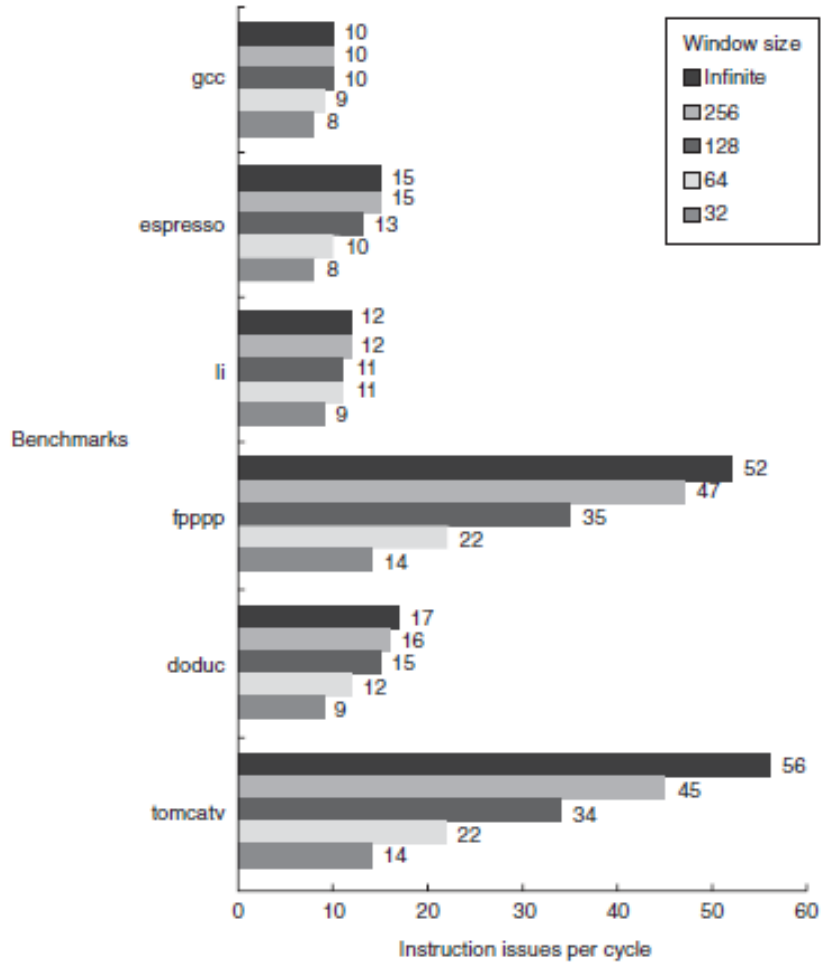

Figure 3.7 The amount of parallelism available versus the window size for a variety of integer and floating-point programs with up to 64 arbitrary instruction issues per clock. Although there are fewer renaming registers than the window size, the fact that all operations have zero latency, and that the number of renaming registers equals the issue width, allows the processor to exploit parallelism within the entire window. In a real implementation, the window size and the number of renaming registers must be balanced to prevent one of these factors from overly constraining the issue rate.

- Figure 3.7 shows the result of the configuration as we vary the window size.
- This configuration is more complex and expensive than any existing implementations, especially in terms of the number of instruction issues, which is more than 10 times larger than the largest number of issues available on any processor in 2005.
- Nonetheless, it gives a useful bound on what future implementations might yield.

Example

- Given the difficulty of increasing the instruction rates with realistic hardware designs, designers face a challenge in deciding *how best to use the limited resources available on an integrated circuit*.
- One of the most interesting trade-offs is between simpler processors with larger caches and higher clock rates versus more emphasis on instruction-level parallelism with a slower clock and smaller caches.
- The following example illustrates the challenges.

Example

Consider the following three hypothetical, but not atypical, processors, which we run with the SPEC gcc benchmark:

1. A simple MIPS two-issue static pipe running at a clock rate of 4 GHz and achieving a pipeline CPI of 0.8. This processor has a cache system that yields 0.005 misses per instruction.
2. A deeply pipelined version of a two-issue MIPS processor with slightly smaller caches and a 5 GHz clock rate. The pipeline CPI of the processor is 1.0, and the smaller caches yield 0.0055 misses per instruction on average.
3. A speculative superscalar with a 64-entry window. It achieves one-half of the ideal issue rate measured for this window size. This processor has the smallest caches, which lead to 0.01 misses per instruction, but it hides 25% of the miss penalty on every miss by dynamic scheduling. This processor has a 2.5 GHz clock.

Example

- Assume that the main memory time (which sets the miss penalty) is 50 ns. Determine the relative performance of these three processors.
- First, we use the miss penalty and miss rate information to compute the contribution to CPI from cache misses for each configuration.
- We do this with the following formula:

  Cache CPI = Misses per instruction x Miss penalty

- We need to compute the miss penalties for each system:

$$\text{Miss penalty} = \frac{\text{Memory access time}}{\text{Clock cycle}}$$

Example

- The clock cycle times for the processors are 250 ps, 200 ps, and 400 ps, respectively.
- Hence, the miss penalties are:

$$\text{Miss penalty}_1 = \frac{50 \text{ ns}}{250 \text{ ps}} = 200 \text{ cycles}$$

$$\text{Miss penalty}_2 = \frac{50 \text{ ns}}{200 \text{ ps}} = 250 \text{ cycles}$$

$$\text{Miss penalty}_3 = \frac{0.75 \times 50 \text{ ns}}{400 \text{ ps}} = 94 \text{ cycles}$$

- Applying this for each cache:

  Cache CPI1 = 0.005 x 200 = 1.0

  Cache CPI2 = 0.0055 x 250 = 1.4

  Cache CPI3 = 0.01 x 94 = 0.94

Example

- We know the pipeline CPI contribution for everything but processor 3; its pipeline CPI is given by:

$$\text{Pipeline CPI}_3 = \frac{1}{\text{Issue rate}} = \frac{1}{9 \times 0.5} = \frac{1}{4.5} = 0.22$$

- Now we can find the CPI for each processor by adding the pipeline and cache CPI contributions:

CPI1 = 0.8 + 1.0 = 1.8

CPI2 = 1.0 + 1.4 = 2.4

CPI3 = 0.22 + 0.94 = 1.16

Example

- Since this is the same architecture, we can compare instruction execution rates in millions of instructions per second (MIPS) to determine relative performance:

$$\text{Instruction execution rate} = \frac{\text{CR}}{\text{CPI}}$$

$$\text{Instruction execution rate}_1 = \frac{4000 \text{ MHz}}{1.8} = 2222 \text{ MIPS}$$

$$\text{Instruction execution rate}_2 = \frac{5000 \text{ MHz}}{2.4} = 2083 \text{ MIPS}$$

$$\text{Instruction execution rate}_3 = \frac{2500 \text{ MHz}}{1.16} = 2155 \text{ MIPS}$$

- The simple two-issue static superscalar looks best. In practice, performance depends on both the CPI and clock rate assumptions.

Outline

- Introduction
- Studies of the Limitations of ILP
- Limitations on ILP for Realizable Processors
- Crosscutting Issues: Hardware versus Software Speculation
- Multithreading: Using ILP Support to Exploit Thread-Level Parallelism
- Putting It All Together: Performance and Efficiency in Advanced Multiple-Issue Processors
- Fallacies and Pitfalls

Hardware versus Software Speculation

- The hardware-intensive approaches to speculation in the previous Lecture provide alternative approaches to exploiting ILP. Some of the trade-offs, and the limitations, for these approaches are listed below:
- To speculate extensively, we must be able to disambiguate memory references.
  - This capability is difficult to do at compile time for integer programs that contain pointers.
  - In a hardware-based scheme, dynamic run time disambiguation of memory addresses is done using the techniques we saw earlier for Tomasulo's algorithm.

Hardware versus Software Speculation

- 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.
  - These properties hold for many integer programs

- Hardware-based speculation maintains a completely precise exception model even for speculated instructions. Recent software-based approaches have added special support to allow this as well.

- Hardware-based speculation does not require compensation or bookkeeping code, which is needed by ambitious software speculation mechanisms.

Hardware versus Software 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 for different implementations of an architecture.
  - Although this advantage is the hardest to quantify, it may be the most important in the long run.

Hardware versus Software Speculation

- The major disadvantage of supporting speculation in hardware is the complexity and additional hardware resources required.

- This hardware cost must be evaluated against both the complexity of a compiler for a software-based approach and the amount and usefulness of the simplifications in a processor that relies on such a compiler.

Hardware versus Software Speculation

- Some designers have tried to combine the dynamic and compiler-based approaches to achieve the best of each.
- Such a combination can generate interesting and obscure interactions.
- For example, if conditional moves are combined with register renaming, subtle side effect appear.
- These subtle interactions complicate the design and verification process and can also reduce performance.

Outline

- Introduction
- Studies of the Limitations of ILP
- Limitations on ILP for Realizable Processors
- Crosscutting Issues: Hardware versus Software Speculation
- Multithreading: Using ILP Support to Exploit Thread-Level Parallelism
- Putting It All Together: Performance and Efficiency in Advanced Multiple-Issue Processors
- Fallacies and Pitfalls

Programming Level Parallelism

- Although increasing performance by using ILP has the great advantage that it is reasonably transparent to the programmer, as we have seen, ILP can be quite limited or hard to exploit in some applications.
- Furthermore, there may be significant parallelism occurring naturally at a higher level in the application that cannot be exploited with the approaches discussed until now.
- For example, an *online transaction-processing system has natural parallelism among the multiple queries* and updates that are presented by requests.
  - These queries and updates can be processed mostly in parallel, since they are largely independent of one another. Of course, many scientific applications contain natural parallelism since they model the three-dimensional, parallel structure of nature, and that structure can be exploited in a simulation.

Thread-level parallelism

- The higher-level parallelism is called *thread-level parallelism* (TLP) because it is logically structured as separate threads of execution.
- A *Thread* is a separate process with its own instructions and data.
- A thread may represent a process that is part of a parallel program consisting of multiple processes, or it may represent an independent program on its own.
  - Each thread has all the state (instructions, data, PC, register state, and so on) necessary to allow it to execute.
- Unlike instruction-level parallelism, which exploits implicit parallel operations within a loop or straight-line code segment, thread-level parallelism is explicitly represented by the use of multiple threads of execution that are inherently parallel.

Thread-level parallelism

- Thread-level parallelism is an important alternative to instruction-level parallelism primarily because it could be more cost-effective to exploit than instruction-level parallelism.
- There are many important applications where thread-level parallelism occurs naturally, as it does in many server applications.
- In other cases, when software is being written from scratch, expressing the inherent parallelism is easy, as is true in some embedded applications.
- Large, established applications written without parallelism in mind, however, pose a significant challenge and can be extremely costly to rewrite to exploit thread-level parallelism.

Combining ILP with TLP

- Thread-level and instruction-level parallelism exploit two different kinds of parallel structure in a program.
- One natural question to ask is whether it is possible for a processor oriented at instruction-level parallelism to exploit thread-level parallelism.
- The motivation for this question comes from the observation that a data path designed to exploit higher amounts of ILP will find that functional units are often idle because of either stalls or dependences in the code.
- *Could the parallelism among threads be used as a source of independent instructions that might keep the processor busy during stalls?*
- *Could this thread-level parallelism be used to employ the functional units that would otherwise lie idle when insufficient ILP exists?*

Hardware support for Multithreading

- *Multithreading* allows multiple threads to share the functional units of a single processor in an overlapping fashion.
- To permit this sharing, the processor must duplicate the independent state of each thread.
  - For example, a separate copy of the register file, a separate PC, and a separate page table are required for each thread.
- The memory itself can be shared through the virtual memory mechanisms, which already support multiprogramming.
- In addition, the hardware must support the ability to change to a different thread relatively quickly;
  - in particular, a thread switch should be much more efficient than a process switch, which typically requires hundreds to thousands of processor cycles.

Fine-grained multithreading

- There are two main approaches to multithreading.
- *Fine-grained multithreading* switches between threads on each instruction, causing the execution of multiple threads to be interleaved.
- This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that time.
- To make fine-grained multithreading practical, the CPU must be able to switch threads on every clock cycle.
- One key advantage of fine-grained multithreading is that it can hide the throughput losses that arise from both short and long stalls, *since instructions from other threads can be executed when one thread stalls*.
- The primary disadvantage of fine-grained multithreading is that it slows down the execution of the individual threads, since a thread that is ready to execute without stalls will be delayed by instructions from other threads.

Coarse-grained multithreading

- *Coarse-grained multithreading* was invented as an alternative to fine-grained multithreading.
- Coarse-grained multithreading switches threads only on costly stalls, such as level 2 cache misses.
- This change relieves the need to have thread-switching be essentially free and is much less likely to slow the processor down, since instructions from other threads will only be issued when a thread encounters a costly stall.

Coarse-grained multithreading

- Coarse-grained multithreading suffers, however, from a major drawback:

  It is limited in its ability to overcome throughput losses, especially from shorter stalls.
- This limitation arises from the pipeline start-up costs of coarse-grain multithreading.
- Because a CPU with coarse-grained multithreading issues instructions from a single thread, when a stall occurs, the pipeline must be emptied or frozen.
- The new thread that begins executing after the stall must fill the pipeline before instructions will be able to complete.
- Because of this start-up overhead, coarsegrained multithreading is much more useful for reducing the penalty of high-cost stalls, where pipeline refill is negligible compared to the stall time.

Fine-grained multithreading that exploits ILP

- Here we will explore a variation on fine-grained multithreading that enables a superscalar processor to exploit ILP and multithreading in an integrated and efficient fashion.

Simultaneous Multithreading: Converting Thread-Level Parallelism into Instruction-Level Parallelism

- Simultaneous multithreading (SMT) is a variation on multithreading that uses the resources of a multiple-issue, dynamically scheduled processor to exploit TLP at the same time it exploits ILP.
- The key insight that motivates SMT is that modern multiple-issue processors often have more functional unit parallelism available than a single thread can effectively use.
- Furthermore, with register renaming and dynamic scheduling, multiple instructions from independent threads can be issued without regard to the dependences among them;
  - The resolution of the dependences can be handled by the dynamic scheduling capability.

Exploiting Superscalars

- Let us conceptually illustrate the differences in a processor's ability to exploit the resources of a superscalar for the following processor configurations:

- A superscalar with no multithreading support
- A superscalar with coarse-grained multithreading
- A superscalar with fine-grained multithreading
- A superscalar with simultaneous multithreading

Four approaches of superscalars

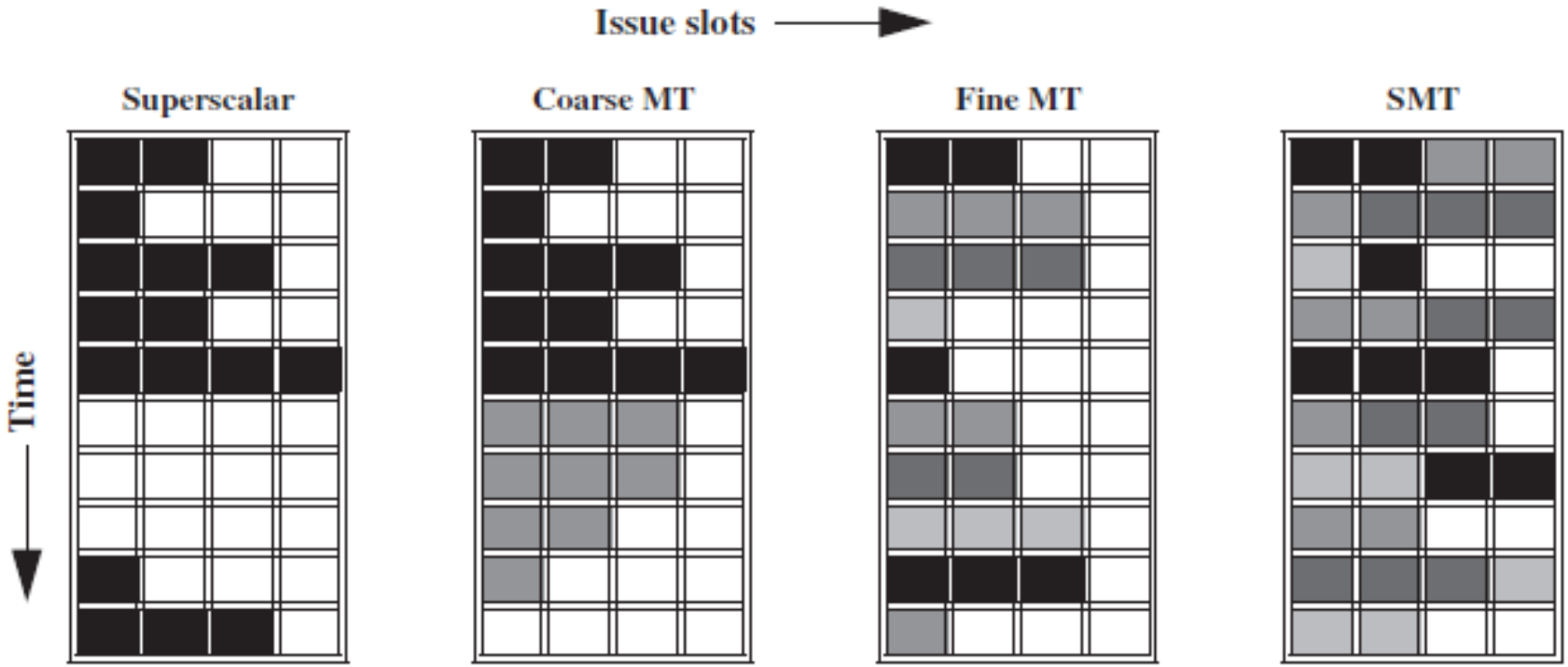

Figure 3.8 How four different approaches use the Issue slots of a superscalar processor. The horizontal dimension represents the instruction issue capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding issue slot is unused in that clock cycle. The shades of grey and black correspond to four different threads in the multithreading processors. Black is also used to indicate the occupied issue slots in the case of the superscalar without multithreading support. The Sun T1 (aka Niagara) processor, which is discussed in the next chapter, is a fine-grained multithreaded architecture.

Design Challenges in SMT

- Because a dynamically scheduled superscalar processor is likely to have a deep pipeline, SMT will be unlikely to gain much in performance if it were coarsegrained.
- Since SMT makes sense only in a fine-grained implementation, we must worry about the impact of fine-grained scheduling on single-thread performance.
- This effect can be minimized by having a preferred thread, which still permits multithreading to preserve some of its performance advantage with a smaller compromise in single-thread performance.

Preferred-thread approach

- At first glance, it might appear that a preferred-thread approach sacrifices neither throughput nor single-thread performance.
- Unfortunately, with a preferred thread, the processor is likely to *sacrifice some throughput when the preferred thread encounters a stall*.
- The reason is that the pipeline is less likely to have a mix of instructions from several threads, resulting in greater probability that either empty slots or a stall will occur.

  Throughput is maximized by having a sufficient number of independent threads to hide all stalls in any combination of threads.

Mixing Threads

- Unfortunately, mixing many threads will inevitably compromise the execution time of individual threads.
- Similar problems exist in instruction fetch.
- To maximize single-thread performance, we should fetch as far ahead as possible in that single thread and always have the fetch unit free when a branch is mispredicted and a miss occurs in the prefetch buffer.

  Unfortunately, this limits the number of instructions available for scheduling from other threads, reducing throughput.
- All multithreaded processors must seek to balance this trade-off.

Mixing Threads

- In practice, the problems of dividing resources and balancing single-thread and multiple-thread performance turn out not to be as challenging as they sound, at least for current superscalar back ends.
- In particular, for current machines that issue four to eight instructions per cycle, it probably suffices to have a small number of active threads, and an even smaller number of “preferred” threads.
- Whenever possible, the processor acts on behalf of a preferred thread.
  - This starts with prefetching instructions: *whenever the prefetch buffers for the preferred threads are not full, instructions are fetched for those threads.*
- Only when the preferred thread buffers are full is the instruction unit directed to prefetch for other threads.

Design challenges for an SMT

Some important design challenges:

- Dealing with a larger register file needed to hold multiple contexts.
- Not affecting the clock cycle, particularly in critical steps such as instruction issue, where more candidate instructions need to be considered, and in instruction completion, where choosing what instructions to commit may be challenging.
- Ensuring that the cache and TLB conflicts generated by the simultaneous execution of multiple threads do not cause significant performance degradation.

Design Observations

Two observations about design are important.

- First, in many cases, *the potential performance overhead due to multithreading is small*, and simple choices work well enough.
- Second, the efficiency of current superscalars is low enough that *there is room for significant improvement*, even at the cost of some overhead.

IBM Power5 Vs. Power4

- The IBM Power5 used the same pipeline as the Power4, but it added SMT support.
- In adding SMT, the designers found that they had to increase a number of structures in the processor so as to minimize the negative performance consequences from fine-grained thread interaction. These changes included the following:
  - Increasing the associativity of the L1 instruction cache and the instruction address translation buffers
  - Adding per-thread load and store queues
  - Increasing the size of the L2 and L3 caches
  - Adding separate instruction prefetch and buffering
  - Increasing the number of virtual registers from 152 to 240
  - Increasing the size of several issue queues

Potential Performance Advantages from SMT

- How much performance can be gained by implementing SMT?
- When this question was explored in 2000–2001, researchers assumed that dynamic superscalars would get much wider in the next five years, supporting six to eight issues per clock with speculative dynamic scheduling, many simultaneous loads and stores, large primary caches, and four to eight contexts with simultaneous fetching from multiple contexts.
- For a variety of reasons, which will become more clear in the next slides, *no processor of this capability has been built nor is likely to be built in the near future*.

IBM Power5

- The IBM Power5 is the most aggressive implementation of SMT as of 2005 and has extensive additions to support SMT, as described in the previous subsection.
- A direct performance comparison of the Power5 in SMT mode, running two copies of an application on a processor, versus the Power5 in single-thread mode, with one process per core, shows speedup across a wide variety of benchmarks of between 0.89 (a performance loss) to 1.41.
- Most applications, however, showed at least some gain from SMT;
  - floating-point-intensive applications, which suffered the most cache conflicts, showed the least gains.

Comparison of SMT and ST

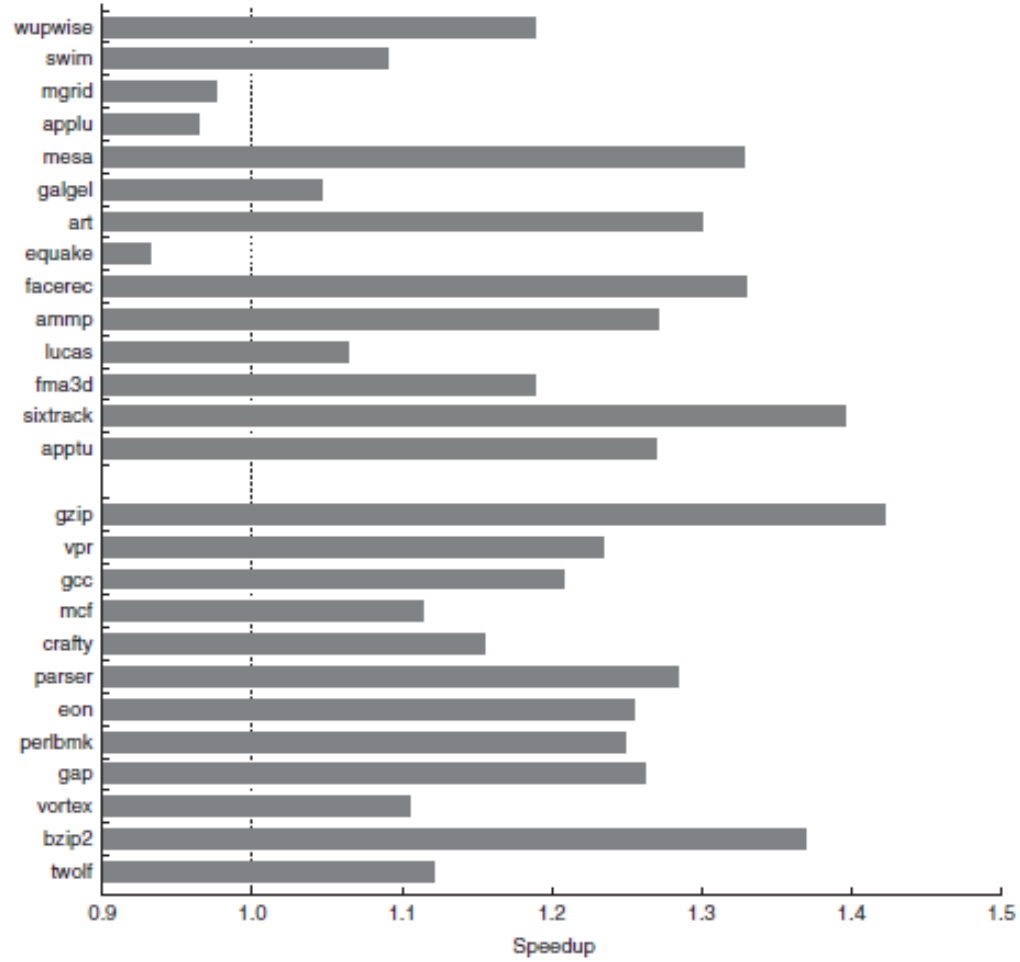

Figure 3.9 A comparison of SMT and single-thread (ST) performance on the 8-processor IBM eServer p5 575. Note that the *y*-axis starts at a speedup of 0.9, a performance loss. Only one processor in each Power5 core is active, which should slightly improve the results from SMT by decreasing destructive interference in the memory system. The SMT results are obtained by creating 16 user threads, while the ST results use only 8 threads; with only one thread per processor, the Power5 is switched to single-threaded mode by the OS. These results were collected by John McCalpin of IBM. As we can see from the data, the standard deviation of the results for the SPECfpRate is higher than for SPECintRate (0.13 versus 0.07), indicating that the SMT improvement for FP programs is likely to vary widely.

Conclusions about SMT

- *The results clearly show the benefit of SMT for an aggressive speculative processor with extensive support for SMT.*
- Because of the costs and diminishing returns in performance, however, rather than implement wider superscalars and more aggressive versions of SMT, many designers are opting to implement multiple CPU cores on a single die with slightly less aggressive support for multiple issue and multithreading.
  - We return to this in the next lecture!

Outline

- Introduction
- Studies of the Limitations of ILP
- Limitations on ILP for Realizable Processors
- Crosscutting Issues: Hardware versus Software Speculation
- Multithreading: Using ILP Support to Exploit Thread-Level Parallelism
- Putting It All Together: Performance and Efficiency in Advanced Multiple-Issue Processors
- Fallacies and Pitfalls

Putting It All Together: Performance and Efficiency in Advanced Multiple-Issue Processors

- In this section, we discuss the characteristics of several recent multiple-issue processors and examine their performance and their efficiency in use of silicon, transistors, and energy.
- We then turn to a discussion of the practical limits of superscalars and the future of high-performance microprocessors.

Four recent multiple-issue processors

Processor	Microarchitecture	Fetch/ issue/ execute	Func. units	Clock rate (GHz)	Transistors and die size	Power
Intel Pentium 4 Extreme	Speculative dynamically scheduled; deeply pipelined; SMT	3/3/4	7 int. 1 FP	3.8	125M 122 mm^2	115 W
AMD Athlon 64 FX-57	Speculative dynamically scheduled	3/3/4	6 int. 3 FP	2.8	114M 115 mm^2	104 W
IBM Power5 1 processor	Speculative dynamically scheduled; SMT; two CPU cores/chip	8/4/8	6 int. 2 FP	1.9	200M 300 mm^2 (estimated)	80 W (estimated)
Intel Itanium 2	EPIC style; primarily statically scheduled	6/5/11	9 int. 2 FP	1.6	592M 423 mm^2	130 W

Figure 3.10 The characteristics of four recent multiple-issue processors. The Power5 includes two CPU cores, although we only look at the performance of one core in this chapter. The transistor count, area, and power consumption of the Power5 are estimated for one core based on two-core measurements of 276M, 389 mm^2, and 125 W, respectively. The large die and transistor count for the Itanium 2 is partly driven by a 9 MB tertiary cache on the chip. The AMD Opteron and Athlon both share the same core microarchitecture. Athlon is intended for desktops and does not support multiprocessing; Opteron is intended for servers and does. This is similar to the differentiation between Pentium and Xeon in the Intel product line.

Comparison of multiple-issue processors: INT benchmarks

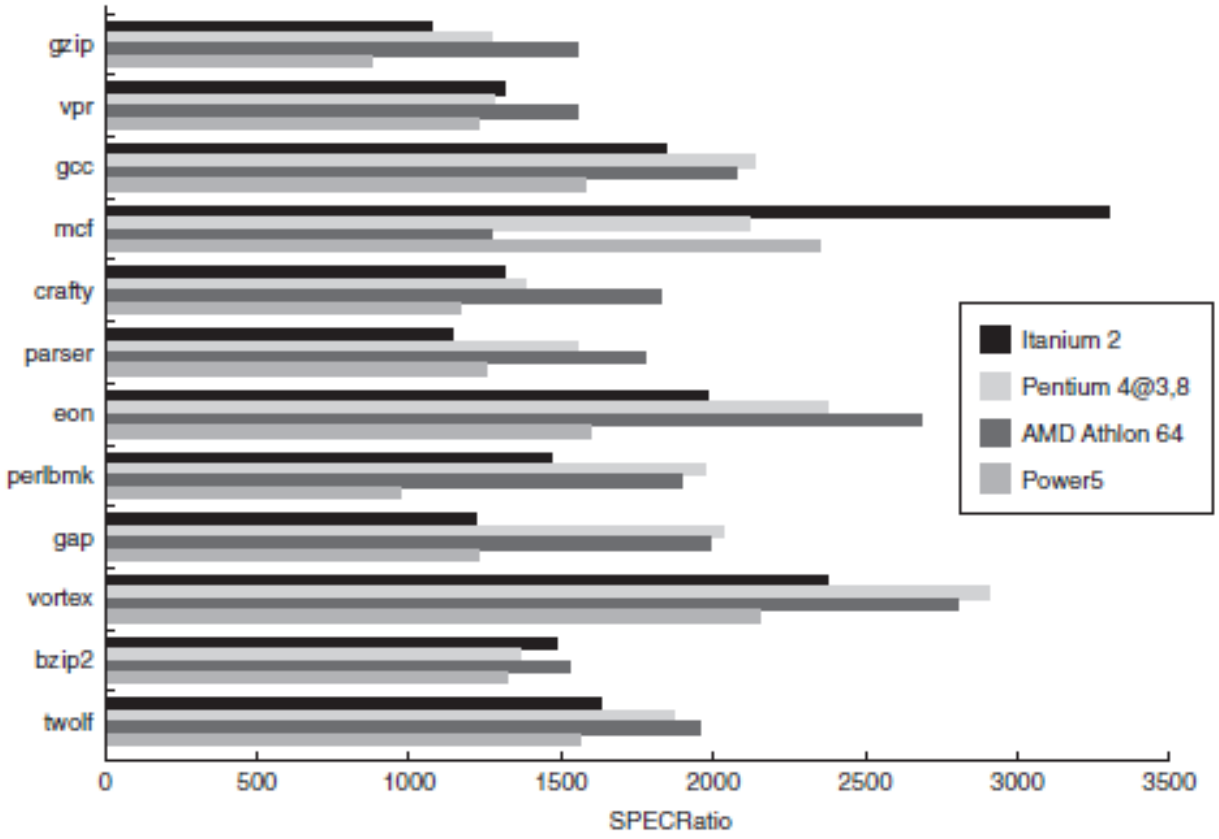

Figure 3.11 A comparison of the performance of the four advanced multiple-issue processors shown in Figure 3.10 for the SPECInt2000 benchmarks.

Comparison of multiple-issue processors: FP benchmarks

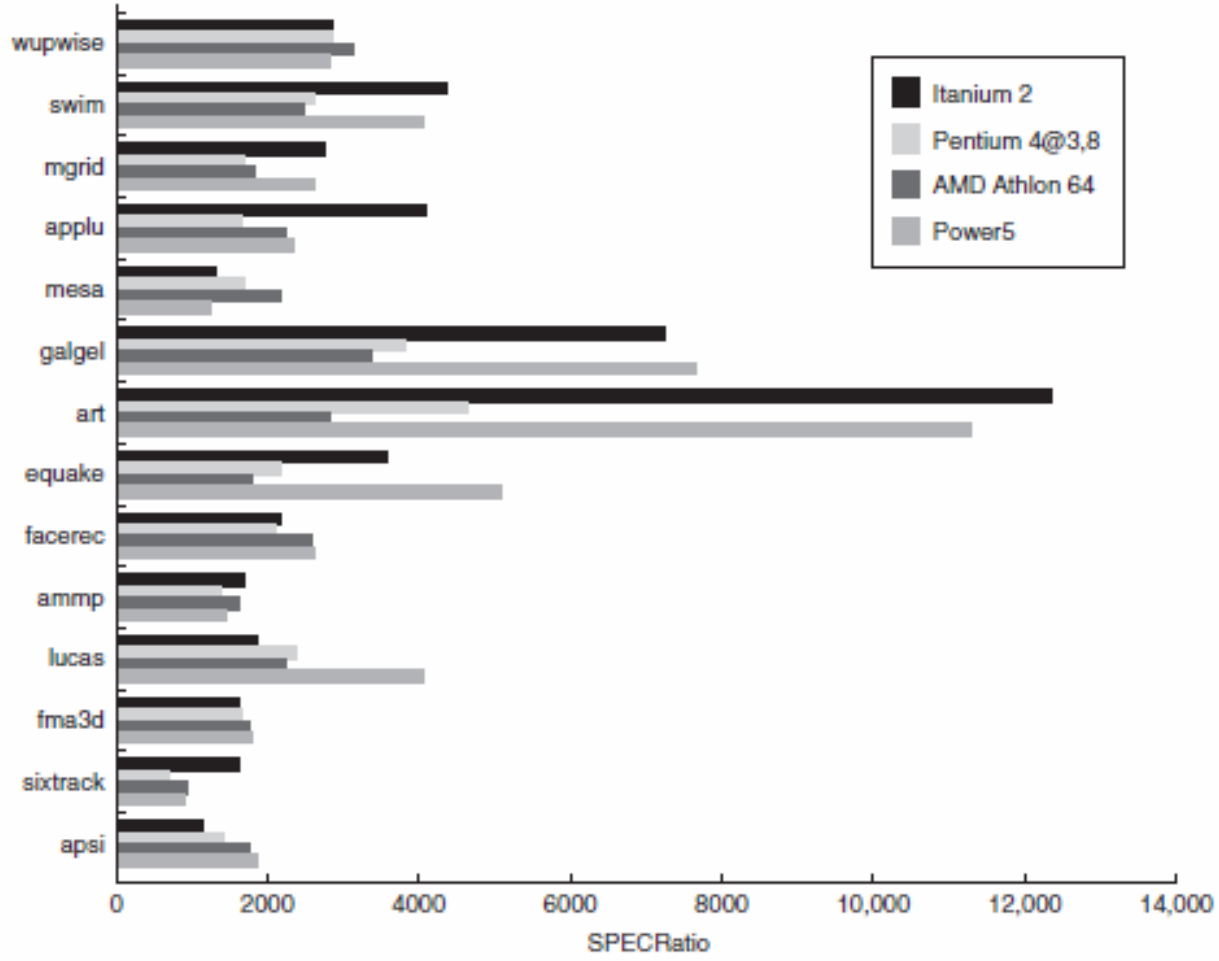

Figure 3.12 A comparison of the performance of the four advanced multiple-issue processors shown in Figure 3.10 for the SPECfp2000 benchmarks.

Efficiency measures for four multiple-issue processors

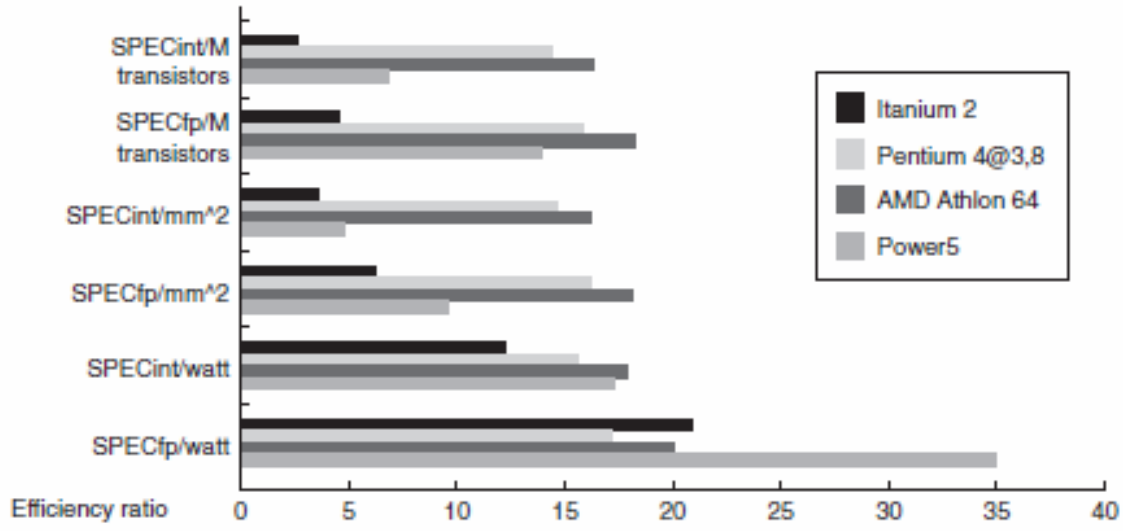

Figure 3.13 Efficiency measures for four multiple-issue processors. In the case of Power5, a single die includes two processor cores, so we estimate the single-core metrics as power = 80 W, area = 290 mm^2, and transistor count = 200M.

What Limits Multiple-Issue Processors?

- Doubling the issue rates above the current rates of 3–6 instructions per clock, say, to 6–12 instructions, will probably require a processor to issue three or four data memory accesses per cycle, resolve two or three branches per cycle, rename and access more than 20 registers per cycle, and fetch 12–24 instructions per cycle.
- The complexities of implementing these capabilities is likely to mean sacrifices in the maximum clock rate.
- For example, the widest-issue processor in Figure 3.10 is the Itanium 2, but it also has the slowest clock rate, despite the fact that it consumes the most power!

What Limits Multiple-Issue Processors? Power issues

- It is now widely accepted that modern microprocessors are primarily power limited.
- Power is a function of both static power, which grows proportionally to the transistor count (whether or not the transistors are switching), and dynamic power, which is proportional to the product of the number of switching transistors and the switching rate.
- Although static power is certainly a design concern, when operating, dynamic power is usually the dominant energy consumer.
- A microprocessor trying to achieve both a low CPI and a high CR must switch more transistors and switch them faster, increasing the power consumption as the product of the two.

Energy Efficient Techniques

- Of course, most techniques for increasing performance, including multiple cores and multithreading, will increase power consumption.
- The key question is whether a technique is *energy efficient:*
  - *Does it increase power consumption faster than it increases performance?*
- Unfortunately, the techniques we currently have to boost the performance of multiple-issue processors all have this inefficiency, which arises from two primary characteristics:
  => next slide

Energy Inefficiency Reasons

- First, issuing multiple instructions incurs some overhead in logic that grows faster than the issue rate grows.
- This logic is responsible for instruction issue analysis, including dependence checking, register renaming, and similar functions.
- The combined result is that, without voltage reductions to decrease power, lower CPIs are likely to lead to lower ratios of performance per watt, simply due to overhead.

Energy Inefficiency Reasons

- Second, and more important, is the growing gap between peak issue rates and sustained performance.
- Since the number of transistors switching will be proportional to the peak issue rate, and the performance is proportional to the sustained rate, a growing performance gap between peak and sustained performance translates to increasing energy per unit of performance.

Speculation and energy efficiency

- Furthermore, the most important technique of the last decade for increasing the exploitation of ILP — namely, speculation — is inherently inefficient. Why?
- Because it can never be perfect; that is, there is inherently waste in executing computations before we know whether they advance the program.
- If speculation were perfect, it could save power, since it would reduce the execution time and save static power, while adding some additional overhead to implement.
- When speculation is not perfect, it rapidly becomes energy inefficient, *since it requires additional dynamic power both for the incorrect speculation and for the resetting of the processor state*.
- Because of the overhead of implementing speculation — register renaming, reorder buffers, more registers, and so on — it is unlikely that any speculative processor could save energy for a significant range of realistic programs.

Clock rate and energy efficiency

- What about focusing on improving clock rate?
- Unfortunately, a similar problem applies to attempts to increase clock rate:

  Increasing the clock rate will increase transistor switching frequency and directly increase power consumption.
- To achieve a faster clock rate, we would need to increase pipeline depth.
- Deeper pipelines, however, incur additional overhead penalties as well as causing higher switching rates.

Outline

- Introduction
- Studies of the Limitations of ILP
- Limitations on ILP for Realizable Processors
- Crosscutting Issues: Hardware versus Software Speculation
- Multithreading: Using ILP Support to Exploit Thread-Level Parallelism
- Putting It All Together: Performance and Efficiency in Advanced Multiple-Issue Processors
- Fallacies and Pitfalls

Fallacy

- *There is a simple approach to multiple-issue processors that yields high performance without a significant investment in silicon area or design complexity.*
- What has been surprising is that many designers have believed that this fallacy was accurate and committed significant effort to trying to find this “silver bullet” approach.
- Although it is possible to build relatively simple multiple-issue processors, as issue rates increase, diminishing returns appear and the silicon and energy costs of wider issue dominate the performance gains.

Pitfall

- *Improving only one aspect of a multiple-issue processor and expecting overall performance improvement.*
- This pitfall is simply a restatement of Amdahl's Law.
- A designer might simply look at a design, see a poor branch-prediction mechanism, and improve it, expecting to see significant performance improvements.
- The difficulty is that many factors limit the performance of multiple-issue machines, and *improving one aspect of a processor often exposes some other aspect that previously did not limit performance*.

End of Lecture 3

- Readings
  - Book: Chapter 3