3

Limits on Instruction-Level Parallelism

Processors are being produced with the potential for very many parallel operations on the instruction level.... Far greater extremes in instruction-level parallelism are on the horizon.

J. Fisher
(1981), in the paper that inaugurated the term "instruction-level parallelism"

One of the surprises about IA-64 is that we hear no claims of high frequency, despite claims that an EPIC processor is less complex than a superscalar processor. It's hard to know why this is so, but one can speculate that the overall complexity involved in focusing on CPI, as IA-64 does, makes it hard to get high megahertz.

M. Hopkins
(2000), in a commentary on the IA-64 architecture, a joint development of HP and Intel designed to achieve dramatic increases in the exploitation of ILP while retaining a simple architecture, which would allow higher performance

3.1 Introduction

As we indicated in the last chapter, 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 close of the last chapter, this era of advances in exploiting ILP appears to be coming to an end.

In this chapter we begin by examining 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. In Section 3.5, we examine the use of thread-level parallelism as an alternative or addition to instruction-level parallelism. Finally, we conclude the chapter by comparing a set of recent processors both in performance and in efficiency measures per transistor and per watt.

3.2 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. The data in this section also provide us with a way to examine the value of ideas that we have introduced in the last chapter, including memory disambiguation, register renaming, and speculation.

In this section we review one of the studies done of these questions. The historical perspectives section in Appendix K describes several studies, including the source for the data in this section (Wall's 1993 study). All these studies of available parallelism operate by making a set of assumptions and seeing how much parallelism is available under those assumptions. The data we examine here are 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.

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 discuss in Section 3.6, we are likely reaching the limits of how much ILP can be exploited efficiently. This section will lay the groundwork to understand why this is the case.

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.
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.

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.

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 instructions in execution at any point are often referred to as *in flight*.)

Of course, this 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.

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.

Figure 3.1 shows the average amount of parallelism available for six of the SPEC92 benchmarks. Throughout this section the parallelism is measured by the average instruction issue rate. Remember that all instructions have a 1-cycle latency; a longer latency would reduce the average number of instructions per clock. Three of these benchmarks (fpppp, doduc, and tomcatv) are floating-point intensive, and the other three are integer programs. Two of the floating-point benchmarks (fpppp and tomcatv) have extensive parallelism, which could be exploited by a vector computer or by a multiprocessor (the structure in fpppp is quite messy, however, since some hand transformations have been done on the code). The doduc program has extensive parallelism, but the parallelism does not occur in simple parallel loops as it does in fpppp and tomcatv. The program li is a LISP interpreter that has many short dependences.

In the next few sections, we 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.

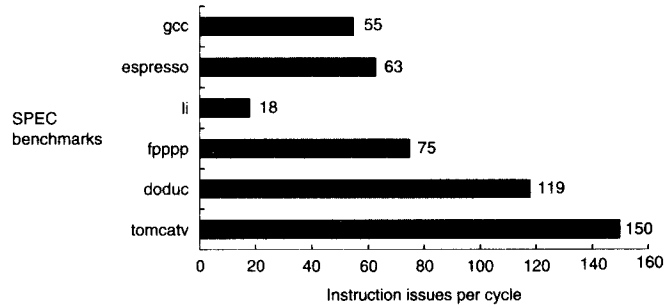

Figure 3.1 ILP available in a perfect processor for six of the SPEC92 benchmarks. The first three programs are integer programs, and the last three are floating-point programs. The floating-point programs are loop-intensive and have large amounts of loop-level parallelism.

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.

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

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

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

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.

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 (today up to $6 \times 200 \times 2 = 2400$), 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. Remember that even though existing processors allow hundreds of instructions to be in flight, because they cannot issue and rename more than a handful in any clock cycle, the maximum throughout is likely to be limited by the issue rate. For example, if the instruction stream contained totally independent instructions that all hit in the cache, a large window would simply never fill. The value of having a window larger than the issue rate occurs when there are dependences or cache misses in the instruction stream.

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.

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, and 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.

Figure 3.2 shows the effects of restricting the size of the window from which an instruction can execute. As we can see in Figure 3.2, the amount of parallelism uncovered falls sharply with decreasing window size. 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 Figure 3.2 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.

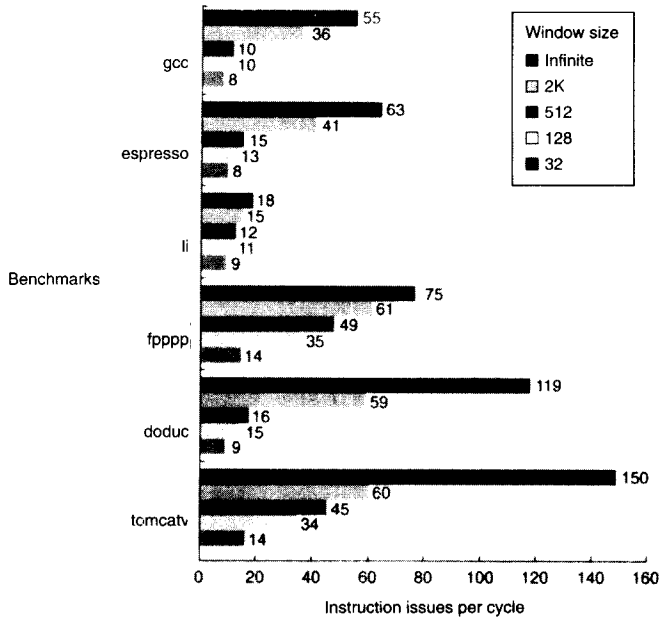

Figure 3.2 The effect of window size shown by each application by plotting the average number of instruction issues per clock cycle.

As we can see in Figure 3.2, the integer programs do not contain nearly as much parallelism as the floating-point programs. This result is to be expected. Looking at how the parallelism drops off in Figure 3.2 makes it clear that the parallelism in the floating-point cases is coming from loop-level parallelism. The fact that the amount of parallelism at low window sizes is not that different among the floating-point and integer programs implies a structure where there are dependences within loop bodies, but few dependences between loop iterations in programs such as tomcatv. At small window sizes, the processors simply cannot see the instructions in the next loop iteration that could be issued in parallel with instructions from the current iteration. This case is an example of where better compiler technology (see Appendix G) could uncover higher amounts of ILP, since it could find the loop-level parallelism and schedule the code to take advantage of it, even with small window sizes.

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 see in the next few sections, 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. Figure 3.3 shows the effects of more realistic prediction schemes in two different formats. 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.

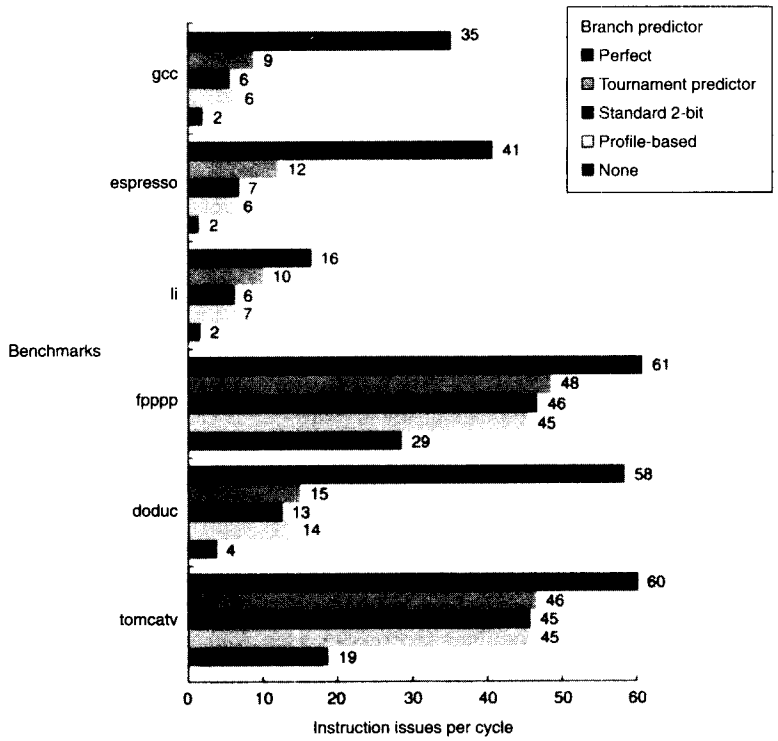

Figure 3.3 The effect of branch-prediction schemes sorted by application. This graph shows the impact of going from a perfect model of branch prediction (all branches predicted correctly arbitrarily far ahead); to various dynamic predictors (selective and 2-bit); to compile time, profile-based prediction; 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).

The five levels of branch prediction shown in these figure 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 2-bit predictor and a noncorrelating 2-bit predictor together with a selector, which chooses the best predictor for each branch. The prediction buffer contains 2^{13} (8K) entries, each consisting of three 2-bit fields, two of which are predictors and the third a selector. The correlating predictor is indexed using the exclusive-or of the branch address and the global branch history. The noncorrelating predictor is the standard 2-bit predictor indexed by the branch address. The selector table is also indexed by the branch address and specifies whether the correlating or noncorrelating predictor should be used. The selector is incremented or decremented just as we would for a standard 2-bit predictor. This predictor, which uses a total of 48K bits, achieves an average misprediction rate of 3% for these six SPEC92 benchmarks and is comparable in strategy and size to the best predictors in use in 2005. Jump prediction is done with a pair of 2K-entry predictors, one organized as a circular buffer for predicting returns and one organized as a standard predictor and used for computed jumps (as in case statements or computed gotos). These jump predictors are nearly perfect.
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.

Since we do *not* charge additional cycles for a mispredicted branch, the only effect of varying the branch prediction is to vary the amount of parallelism that can be exploited across basic blocks by speculation. Figure 3.4 shows the accuracy of the three realistic predictors for the conditional branches for the subset of SPEC92 benchmarks we include here.

Figure 3.3 shows that the branch behavior of two of the floating-point programs is much simpler than the other programs, primarily because these two programs have many fewer branches and the few branches that exist are more predictable. This property allows significant amounts of parallelism to be exploited with realistic prediction schemes. In contrast, for all the integer programs and for doduc, the FP benchmark with the least loop-level parallelism, even the difference between perfect branch prediction and the ambitious selective predictor is dramatic. Like the window size data, these figures tell us that to achieve significant amounts of parallelism in integer programs, the processor must select and execute instructions that are widely separated. When branch

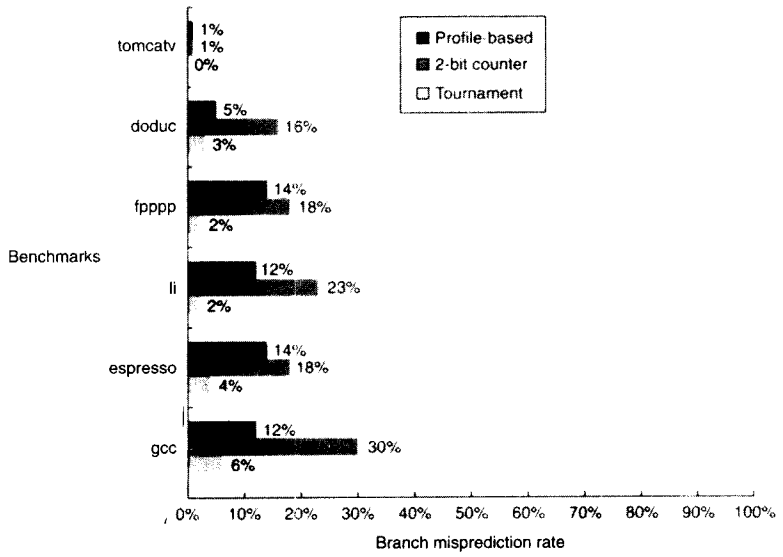

Figure 3.4 Branch misprediction rate for the conditional branches in the SPEC92 subset.

prediction is not highly accurate, the mispredicted branches become a barrier to finding the parallelism.

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 (see Section 3.5), and all are available to a single thread when in single-thread mode. Figure 3.5 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.

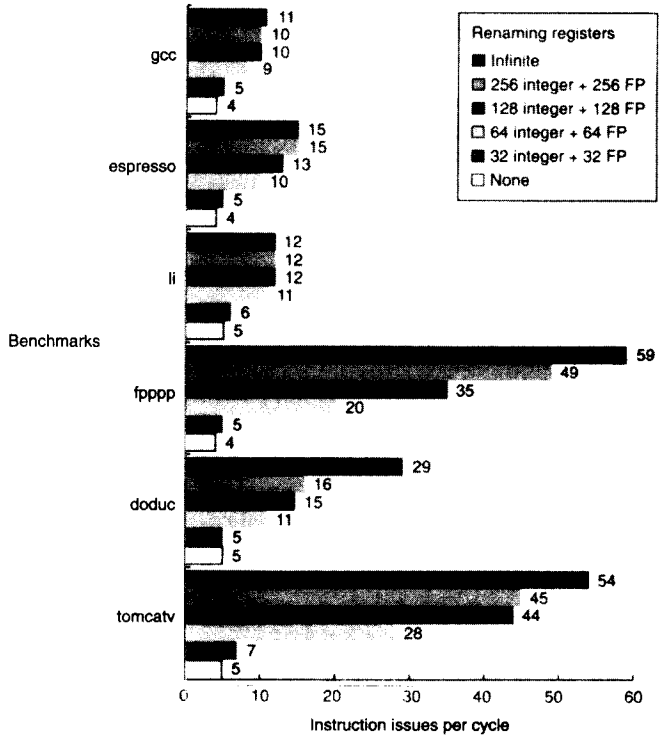

Figure 3.5 The reduction in available parallelism is significant when fewer than an unbounded number of renaming registers are available. *Both* the number of FP registers and the number of GP registers are increased by the number shown on the *x*-axis. So, the entry corresponding to "128 integer + 128 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.

The results in this figure might seem somewhat surprising: You might expect that name dependences should only slightly reduce the parallelism available. Remember though, that exploiting large amounts of parallelism requires evaluating many possible execution paths, speculatively. Thus, many registers are needed to hold live variables from these threads. Figure 3.5 shows that the impact of having only a finite number of registers is significant if extensive parallelism exists. Although this graph shows 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 comparable to the number of extra registers available on any existing processor as of 2005.

Although register renaming is obviously critical to performance, an infinite number of registers is 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 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). Figure 3.6 shows 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 repre-

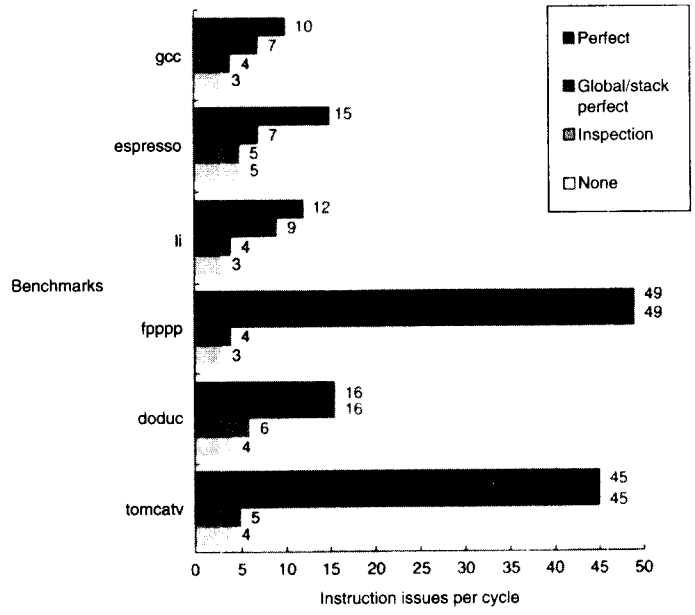

Figure 3.6 The effect of varying levels of alias analysis on individual programs. Anything less than perfect analysis has a dramatic impact on the amount of parallelism found in the integer programs, and global/stack analysis is perfect (and unrealizable) for the FORTRAN programs.

sents 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, assuming R10 could not have changed. 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.

As you might expect, for the FORTRAN programs (where no heap references exist), there is no difference between perfect and global/stack perfect analysis. The global/stack perfect analysis is optimistic, since no compiler could ever find all array dependences exactly. The fact that perfect analysis of global and stack references is still a factor of two better than inspection indicates that either sophisticated compiler analysis or dynamic analysis on the fly will be required to obtain much parallelism. In practice, dynamically scheduled processors rely on dynamic memory disambiguation. To implement perfect dynamic disambiguation for a given load, we must know the memory addresses of all earlier stores that have not yet committed, since a load may have a dependence through memory on a store. As we mentioned in the last chapter, memory address speculation could be used to overcome this limit.

3.3 Limitations on ILP for Realizable Processors

In this section 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.

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.

Figure 3.7 shows the result for this 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

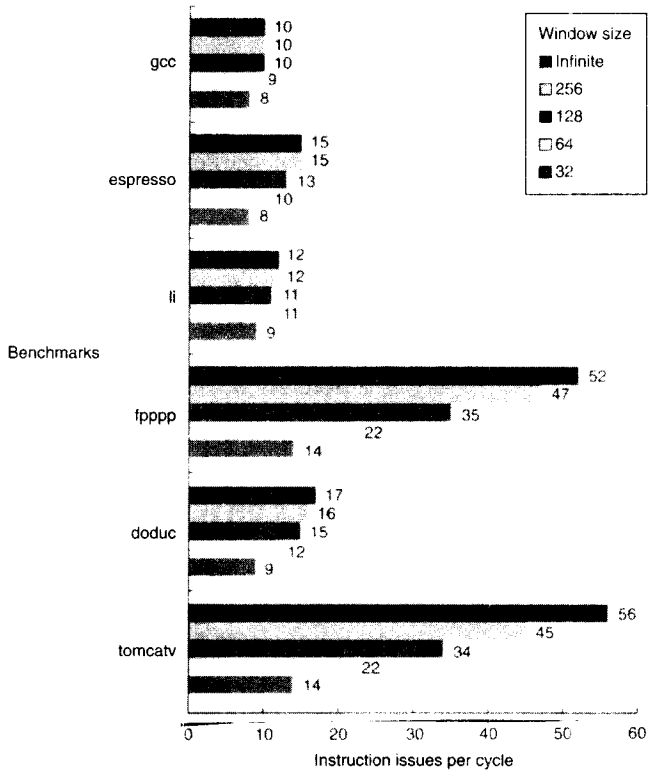

Figure 3.7 The amount of parallelism available versus the window size for a variety of integer and floating-point programs with up to 64 arbitrary instruction issues per clock. Although there are fewer renaming registers than the window size, the fact that all operations have zero latency, and that the 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.

yield. The data in these figures are likely to be very optimistic for another reason. There are no issue restrictions among the 64 instructions: They may all be memory references. No one would even contemplate this capability in a processor in the near future. Unfortunately, it is quite difficult to bound the performance of a processor with reasonable issue restrictions; not only is the space of possibilities quite large, but the existence of issue restrictions requires that the parallelism be evaluated with an accurate instruction scheduler, making the cost of studying processors with large numbers of issues very expensive.

In addition, remember that in interpreting these results, cache misses and nonunit latencies have not been taken into account, and both these effects will have significant impact!

The most startling observation from Figure 3.7 is that with the realistic processor constraints listed above, the effect of the window size for the integer programs is not as 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. Indeed, most of the market growth in the last decade—transaction processing, web servers, and the like—depended on integer performance, rather than floating point. As we will see in the next section, for a realistic processor in 2005, the actual performance levels are much lower than those shown in Figure 3.7.

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. (Use the data in Figure 3.7.)

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.

Assume that the main memory time (which sets the miss penalty) is 50 ns. Determine the relative performance of these three processors.

Answer 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:

$$\text{Cache CPI} = \text{Misses per instruction} \times \text{Miss penalty}$$

We need to compute the miss penalties for each system:

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

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:

$\text{Cache CPI}_1 = 0.005 \times 200 = 1.0$

$\text{Cache CPI}_2 = 0.0055 \times 250 = 1.4$

$\text{Cache CPI}_3 = 0.01 \times 94 = 0.94$

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:

$\text{CPI}_1 = 0.8 + 1.0 = 1.8$

$\text{CPI}_2 = 1.0 + 1.4 = 2.4$

$\text{CPI}_3 = 0.22 + 0.94 = 1.16$

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

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

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

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

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

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

Beyond the Limits of This Study

Like any limit study, the study we have examined in this section has its own limitations. We divide these into two classes: limitations 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. *WAW and WAR 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 [1992] examine 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 Appendix G, 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 et al. [1999] explored the advantages of removing this constraint.

3. *Overcoming the data flow limit*—If value prediction worked with high accuracy, it could overcome the data flow limit. As of yet, none of the more than 50 papers on the subject have achieved a significant enhancement in ILP when using a realistic prediction scheme. Obviously, perfect data value prediction would lead to effectively infinite parallelism, since every value of every instruction could be predicted a priori.

For a less-than-perfect processor, several ideas have been proposed that could expose more ILP. One example is to speculate along multiple paths. This idea was discussed by Lam and Wilson [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 [1993] provides data for speculating in both directions on up to eight branches. Given the costs of pursuing both paths, knowing that one will be thrown away (and the growing amount of useless computation as such a process is followed through multiple branches), every commercial design has instead devoted additional hardware to better speculation on the correct path.

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 Section 3.6, the implications of ILP limitations and the costs of implementing wider issue seem to have created effective limitations on ILP exploitation.

3.4 Crosscutting Issues: Hardware versus Software Speculation

"Crosscutting Issues" is a section that discusses topics that involve subjects from different chapters. The next few chapters include such a section.

The hardware-intensive approaches to speculation in the previous chapter and the software approaches of Appendix G 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 disam-

biguation of memory addresses is done using the techniques we saw earlier for Tomasulo's algorithm. This disambiguation allows us to move loads past stores at run time. Support for speculative memory references can help overcome the conservatism of the compiler, but unless such approaches are used carefully, the overhead of the recovery mechanisms may swamp the advantages.

- 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. For example, a good static predictor has a misprediction rate of about 16% for four major integer SPEC92 programs, and a hardware predictor has a misprediction rate of under 10%. Because speculated instructions may slow down the computation when the prediction is incorrect, this difference is significant. One result of this difference is that even statically scheduled processors normally include dynamic branch predictors.
- 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.
- 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. Interestingly, this was one of the motivations for the IBM 360/91. On the other hand, more recent explicitly parallel architectures, such as IA-64, have added flexibility that reduces the hardware dependence inherent in a code sequence.

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. We return to this topic in the concluding remarks.

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, a subtle side effect appears. A conditional move that is annulled must still copy a value to the destination register, since it was renamed earlier in the instruction pipeline. These subtle interactions complicate the design and verification process and can also reduce performance.

3.5 Multithreading: Using ILP Support to Exploit Thread-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 in this chapter. 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.

This 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 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, the software is being written from scratch, and 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. Chapter 4 explores multiprocessors and the support they provide for thread-level parallelism.

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?

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.

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 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 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, coarse-grained multithreading is much more useful for reducing the penalty of high-cost stalls, where pipeline refill is negligible compared to the stall time.

The next subsection explores a variation on fine-grained multithreading that enables a superscalar processor to exploit ILP and multithreading in an integrated and efficient fashion. In Chapter 4, we return to the issue of multithreading when we discuss its integration with multiple CPUs in a single chip.

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.

Figure 3.8 conceptually illustrates 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

In the superscalar without multithreading support, the use of issue slots is limited by a lack of ILP, a topic we discussed in earlier sections. In addition, a major stall, such as an instruction cache miss, can leave the entire processor idle.

In the coarse-grained multithreaded superscalar, the long stalls are partially hidden by switching to another thread that uses the resources of the processor. Although this reduces the number of completely idle clock cycles, within each clock cycle, the ILP limitations still lead to idle cycles. Furthermore, in a coarse-grained multithreaded processor, since thread switching only occurs when there is a stall and the new thread has a start-up period, there are likely to be some fully idle cycles remaining.

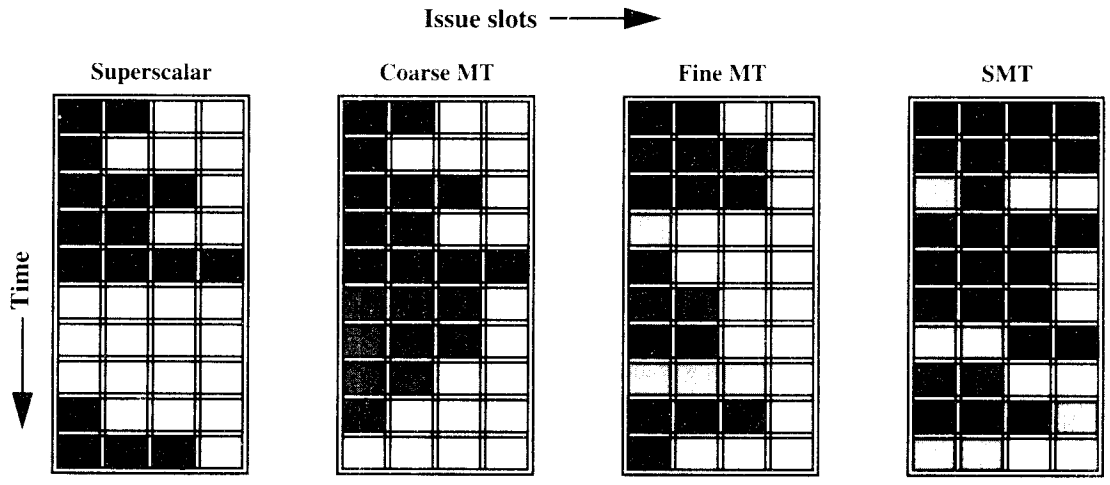

Figure 3.8 How four different approaches use the issue slots of a superscalar processor. The horizontal dimension represents the instruction issue capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding issue 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.

In the fine-grained case, the interleaving of threads eliminates fully empty slots. Because only one thread issues instructions in a given clock cycle, however, ILP limitations still lead to a significant number of idle slots within individual clock cycles.

In the SMT case, TLP and ILP are exploited simultaneously, with multiple threads using the issue slots in a single clock cycle. Ideally, the issue slot usage is limited by imbalances in the resource needs and resource availability over multiple threads. In practice, other factors—including how many active threads are considered, finite limitations on buffers, the ability to fetch enough instructions from multiple threads, and practical limitations of what instruction combinations can issue from one thread and from multiple threads—can also restrict how many slots are used. Although Figure 3.8 greatly simplifies the real operation of these processors, it does illustrate the potential performance advantages of multithreading in general and SMT in particular.

As mentioned earlier, simultaneous multithreading uses the insight that a dynamically scheduled processor already has many of the hardware mechanisms needed to support the integrated exploitation of TLP through multithreading. In particular, dynamically scheduled superscalars have a large set of virtual registers that can be used to hold the register sets of independent threads (assuming separate renaming tables are kept for each thread). Because register renaming provides unique register identifiers, instructions from multiple threads can be mixed in the data path without confusing sources and destinations across the threads.

This observation leads to the insight that multithreading can be built on top of an out-of-order processor by adding a per-thread renaming table, keeping separate PCs, and providing the capability for instructions from multiple threads to commit.

There are complications in handling instruction commit, since we would like instructions from independent threads to be able to commit independently. The independent commitment of instructions from separate threads can be supported by logically keeping a separate reorder buffer for each thread.

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 coarse-grained. 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.

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.

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.

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. Note that having two preferred threads means that we are simultaneously prefetching for two instruction streams, and this adds complexity to the instruction fetch unit and the instruction cache. Similarly, the instruction issue unit can direct its attention to the preferred threads, considering other threads only if the preferred threads are stalled and cannot issue.

There are a variety of other design challenges for an SMT processor, including the following:

- 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

In viewing these problems, two observations 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.

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

Because SMT exploits thread-level parallelism on a multiple-issue superscalar, it is most likely to be included in high-end processors targeted at server markets. In addition, it is likely that there will be some mode to restrict the multithreading, so as to maximize the performance of a single thread.

Potential Performance Advantages from SMT

A key question is, 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 section, no processor of this capability has been built nor is likely to be built in the near future.

As a result, simulation research results that showed gains for multiprogrammed workloads of two or more times are unrealistic. In practice, the existing implementations of SMT offer only two contexts with fetching from only one, as well as more modest issue abilities. The result is that the gain from SMT is also more modest.

For example, in the Pentium 4 Extreme, as implemented in HP-Compaq servers, the use of SMT yields a performance improvement of 1.01 when running the SPECintRate benchmark and about 1.07 when running the SPECfpRate benchmark. In a separate study, Tuck and Tullsen [2003] observe that running a mix of each of the 26 SPEC benchmarks paired with every other SPEC benchmark (that is, 26^2 runs, if a benchmark is also run opposite itself) results in speedups ranging from 0.90 to 1.58, with an average speedup of 1.20. (Note that this measurement is different from SPECRate, which requires that each SPEC benchmark be run against a vendor-selected number of copies of the same benchmark.) On the SPLASH parallel benchmarks, they report multithreaded speedups ranging from 1.02 to 1.67, with an average speedup of about 1.22.

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.

Figure 3.9 shows the speedup for an 8-processor Power5 multiprocessor with and without SMT for the SPECRate2000 benchmarks, as described in the caption. On average, the SPECintRate is 1.23 times faster, while the SPECfpRate is 1.16 times faster. Note that a few floating-point benchmarks experience a slight decrease in performance in SMT mode, with the maximum reduction in speedup being 0.93.

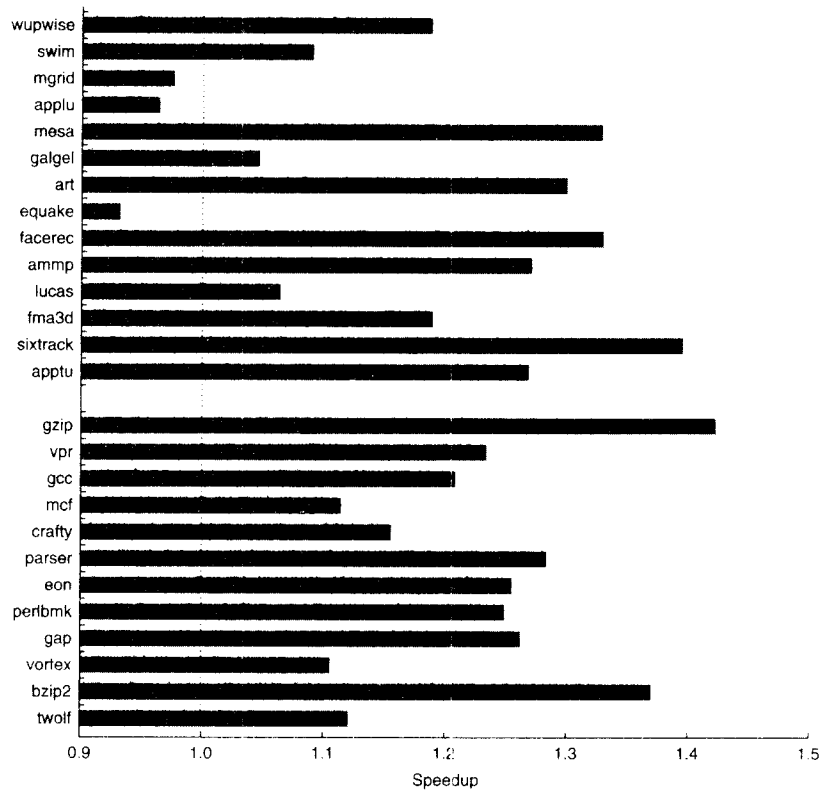

Figure 3.9 A comparison of SMT and single-thread (ST) performance on the 8-processor IBM eServer p5 575. Note that the *y*-axis starts at a speedup of 0.9, a performance loss. Only one processor in each Power5 core is active, which should slightly improve the results from SMT by decreasing destructive 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.

These 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 topic in the next chapter.

3.6 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.

Figure 3.10 shows the characteristics of four of the most recent high-performance microprocessors. They vary widely in organization, issue rate, functional unit capability, clock rate, die size, transistor count, and power. As Figures 3.11 and 3.12 show, there is no obvious overall leader in performance. The Itanium 2 and Power5, which perform similarly on SPECfp, clearly dominate the Athlon and Pentium 4 on those benchmarks. The AMD Athlon leads on SPECint performance followed by the Pentium 4, Itanium 2, and Power5.

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.

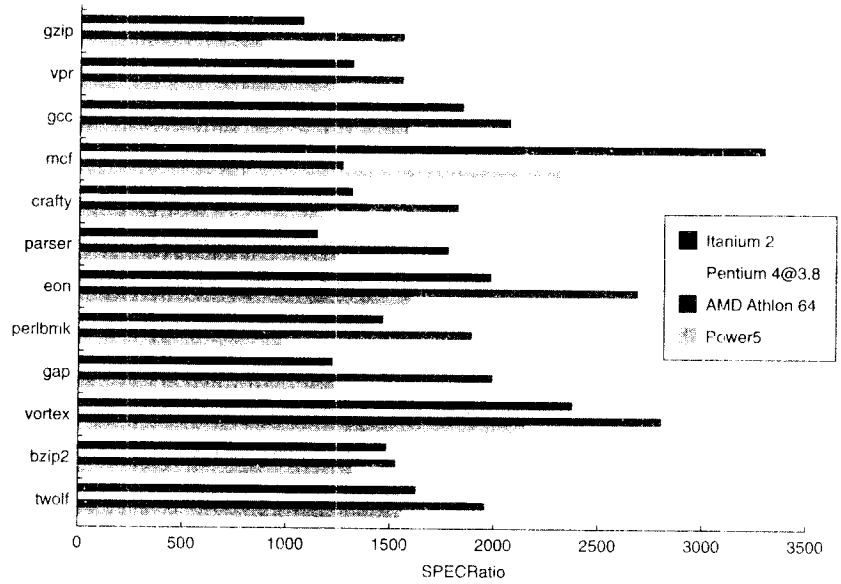

Figure 3.11 A comparison of the performance of the four advanced multiple-issue processors shown in Figure 3.10 for the SPECint2000 benchmarks.

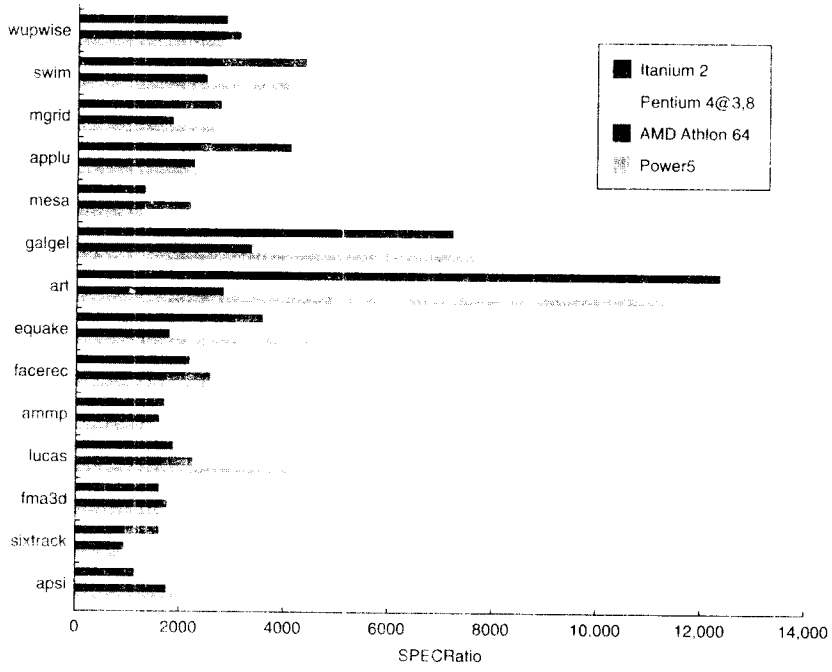

Figure 3.12 A comparison of the performance of the four advanced multiple-issue processors shown in Figure 3.10 for the SPECfp2000 benchmarks.

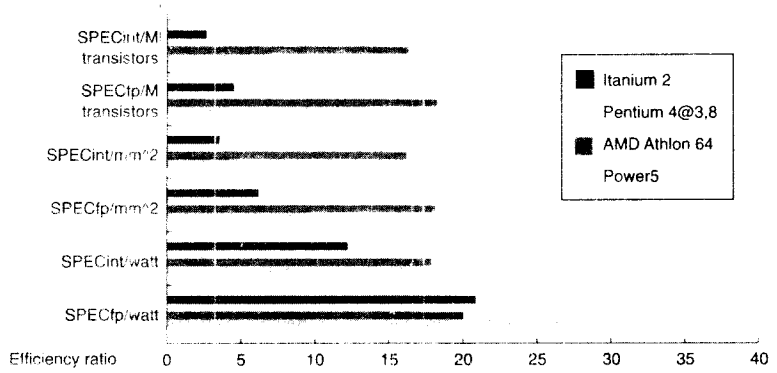

Figure 3.13 Efficiency measures for four multiple-issue processors. In the case of Power5, a single die includes two processor cores, so we estimate the single-core metrics as power = 80 W, area = 290 mm^2, and transistor count = 200M.

As important as overall performance is, the question of efficiency in terms of silicon area and power is equally critical. As we discussed in Chapter 1, power has become the major constraint on modern processors. Figure 3.13 shows how these processors compare in terms of efficiency, by charting the SPECint and SPECfp performance versus the transistor count, silicon area, and power. The results provide an interesting contrast to the performance results. The Itanium 2 is the most inefficient processor both for floating-point and integer code for all but one measure (SPECfp/watt). The Athlon and Pentium 4 both make good use of transistors and area in terms of efficiency, while the IBM Power5 is the most effective user of energy on SPECfp and essentially tied on SPECint. The fact that none of the processors offer an overwhelming advantage in efficiency across multiple measures leads us to believe that none of these approaches provide a "silver bullet" that will allow the exploitation of ILP to scale easily and efficiently much beyond current levels.

Let's try to understand why this is the case.

What Limits Multiple-Issue Processors?

The limitations explored in Sections 3.1 and 3.3 act as significant barriers to exploiting more ILP, but they are not the only barriers. For example, 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!

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.

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.

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.

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. Unfortunately, this growing gap appears to be quite fundamental and arises from many of the issues we discuss in Sections 3.2 and 3.3. For example, if we want to *sustain* four instructions per clock, we must *fetch* more, *issue* more, and *initiate execution* on more than four instructions. The power will be proportional to the peak rate, but performance will be at the sustained rate. (In many recent processors, provision has been made for decreasing power consumption by shutting down an inactive portion of a processor, including powering off the clock to that portion of the chip. Such techniques, while useful, cannot prevent the long-term decrease in power 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.

What about focusing on improving clock rate? Unfortunately, a similar conundrum 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.

The best example of this phenomenon comes from comparing the Pentium III and Pentium 4. To a first approximation, the Pentium 4 is a deeply pipelined version of the Pentium III architecture. In a similar process, it consumes roughly an amount of power proportional to the difference in clock rate. Unfortunately, its performance is somewhat less than the ratio of the clock rates because of overhead and ILP limitations

It appears that we have reached—and, in some cases, possibly even surpassed—the point of diminishing returns in our attempts to exploit ILP. The implications of these limits can be seen over the last few years in the slower performance growth rates (see Chapter 1), in the lack of increase in issue capability, and in the emergence of multicore designs; we return to this issue in the concluding remarks.

3.7 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.*

The last few sections should have made this point obvious. 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.

In addition to the hardware inefficiency, it has become clear that compiling for processors with significant amounts of ILP has become extremely complex. Not only must the compiler support a wide set of sophisticated transformations, but tuning the compiler to achieve good performance across a wide set of benchmarks appears to be very difficult.

Obtaining good performance is also affected by design decisions at the system level, and such choices can be complex, as the last section clearly illustrated.

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.

We can see examples of this in the data on ILP. For example, looking just at the effect of branch prediction in Figure 3.3 on page 160, we can see that going from a standard 2-bit predictor to a tournament predictor significantly improves the parallelism in espresso (from an issue rate of 7 to an issue rate of 12). If the processor provides only 32 registers for renaming, however, the amount of parallelism is limited to 5 issues per clock cycle, even with a branch-prediction scheme better than either alternative.

3.8 Concluding Remarks

The relative merits of software-intensive and hardware-intensive approaches to exploiting ILP continue to be debated, although the debate has shifted in the last five years. Initially, the software-intensive and hardware-intensive approaches were quite different, and the ability to manage the complexity of the hardware-intensive approaches was in doubt. The development of several high-performance dynamic speculation processors, which have high clock rates, has eased this concern.

The complexity of the IA-64 architecture and the Itanium design has signaled to many designers that it is unlikely that a software-intensive approach will produce processors that are significantly faster (especially for integer code), smaller (in transistor count or die size), simpler, or more power efficient. It has become clear in the past five years that the IA-64 architecture does *not* represent a significant breakthrough in scaling ILP or in avoiding the problems of complexity and power consumption in high-performance processors. Appendix H explores this assessment in more detail.

The limits of complexity and diminishing returns for wider issue probably also mean that only limited use of simultaneous multithreading is likely. It simply is not worthwhile to build the very wide issue processors that would justify the most aggressive implementations of SMT. For this reason, existing designs have used modest, two-context versions of SMT or simple multithreading with two contexts, which is the appropriate choice with simple one- or two-issue processors.

Instead of pursuing more ILP, architects are increasingly focusing on TLP implemented with single-chip multiprocessors, which we explore in the next chapter. In 2000, IBM announced the first commercial single-chip, general-purpose multiprocessor, the Power4, which contains two Power3 processors and an integrated second-level cache. Since then, Sun Microsystems, AMD, and Intel

have switched to a focus on single-chip multiprocessors rather than more aggressive uniprocessors.

The question of the right balance of ILP and TLP is still open in 2005, and designers are exploring the full range of options, from simple pipelining with more processors per chip, to aggressive ILP and SMT with fewer processors. It may well be that the right choice for the server market, which can exploit more TLP, may differ from the desktop, where single-thread performance may continue to be a primary requirement. We return to this topic in the next chapter.

3.9 Historical Perspective and References

Section K.4 on the companion CD features a discussion on the development of pipelining and instruction-level parallelism. We provide numerous references for further reading and exploration of these topics.

Case Study with Exercises by Wen-mei W. Hwu and John W. Sias

Concepts illustrated by this case study

- Limited ILP due to software dependences
- Achievable ILP with hardware resource constraints
- Variability of ILP due to software and hardware interaction
- Tradeoffs in ILP techniques at compile time vs. execution time

Case Study: Dependences and Instruction-Level Parallelism

The purpose of this case study is to demonstrate the interaction of hardware and software factors in producing instruction-level parallel execution. This case study presents a concise code example that concretely illustrates the various limits on instruction-level parallelism. By working with this case study, you will gain intuition about how hardware and software factors interact to determine the execution time of a particular type of code on a given system.

A hash table is a popular data structure for organizing a large collection of data items so that one can quickly answer questions such as, "Does an element of value 100 exist in the collection?" This is done by assigning data elements into one of a large number of buckets according to a hash function value generated from the data values. The data items in each bucket are typically organized as a linked list sorted according to a given order. A lookup of the hash table starts by determining the bucket that corresponds to the data value in question. It then traverses the linked list of data elements in the bucket and checks if any element

in the list has the value in question. As long as one keeps the number of data elements in each bucket small, the search result can be determined very quickly.

The C source code in Figure 3.14 inserts a large number (N_ELEMENTS) of elements into a hash table, whose 1024 buckets are all linked lists sorted in ascending order according to the value of the elements. The array element[] contains the elements to be inserted, allocated on the heap. Each iteration of the outer (for) loop, starting at line 6, enters one element into the hash table.

Line 9 in Figure 3.14 calculates hash_index, the hash function value, from the data value stored in element[i]. The hashing function used is a very simple

```
1     typedef struct _Element {
2        int value;
3        struct _Element *next;
4     } Element;
5     Element element[N_ELEMENTS], *bucket[1024];
      /* The array element is initialized with the items to be inserted;
         the pointers in the array bucket are initialized to NULL. */

6     for (i = 0; i < N_ELEMENTS; i++)
       {
7        Element *ptrCurr, **ptrUpdate;
8        int hash_index;

         /* Find the location at which the new element is to be inserted. */
9        hash_index = element[i].value & 1023;
10       ptrUpdate = &bucket[hash_index];
11       ptrCurr = bucket[hash_index];
         /* Find the place in the chain to insert the new element. */
12       while (ptrCurr &&
13              ptrCurr->value <= element[i].value)
14        {
15    ptrUpdate = &ptrCurr->next;
16          ptrCurr = ptrCurr->next;
          }

         /* Update pointers to insert the new element into the chain. */
17       element[i].next = *ptrUpdate;
18       *ptrUpdate = &element[i];
      }
```

Figure 3.14 Hash table code example.

one; it consists of the least significant 10 bits of an element's data value. This is done by computing the bitwise logical AND of the element data value and the (binary) bit mask 11 1111 1111 (1023 in decimal).

Figure 3.15 illustrates the hash table data structure used in our C code example. The `bucket` array on the left side of Figure 3.15 is the hash table. Each entry of the `bucket` array contains a pointer to the linked list that stores the data elements in the bucket. If bucket `i` is currently empty, the corresponding `bucket[i]` entry contains a `NULL` pointer. In Figure 3.15, the first three buckets contain one data element each; the other buckets are empty.

Variable `ptrCurr` contains a pointer used to examine the elements in the linked list of a bucket. At Line 11 of Figure 3.14, `ptrCurr` is set to point to the first element of the linked list stored in the given bucket of the hash table. If the bucket selected by the `hash_index` is empty, the corresponding bucket array entry contains a `NULL` pointer.

The `while` loop starts at line 12. Line 12 tests if there is any more data elements to be examined by checking the contents of variable `ptrCurr`. Lines 13 through 16 will be skipped if there are no more elements to be examined, either because the bucket is empty, or because all the data elements in the linked list have been examined by previous iterations of the `while` loop. In the first case, the new data element will be inserted as the first element in the bucket. In the second case, the new element will be inserted as the last element of the linked list.

In the case where there are still more elements to be examined, line 13 tests if the current linked list element contains a value that is smaller than or equal to that of the data element to be inserted into the hash table. If the condition is true, the `while` loop will continue to move on to the next element in the linked list; lines 15 and 16 advance to the next data element of the linked list by moving `ptrCurr` to the next element in the linked list. Otherwise, it has found the position in the

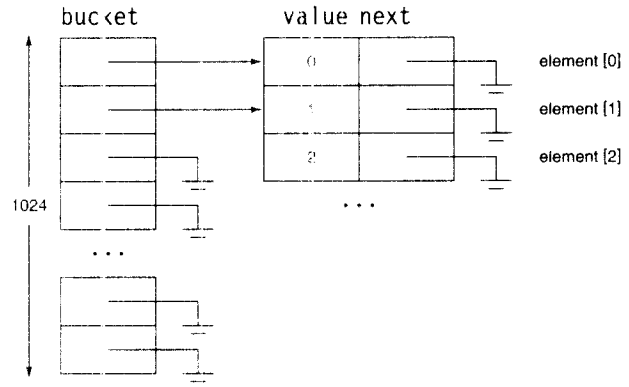

Figure 3.15 Hash table data structure.

linked list where the new data element should be inserted, the `while` loop will terminate and the new data element will be inserted right before the element pointed to by `ptrCurr`.

The variable `ptrUpdate` identifies the pointer that must be updated in order to insert the new data element into the bucket. It is set by line 10 to point to the bucket entry. If the bucket is empty, the `while` loop will be skipped altogether and the new data element is inserted by changing the pointer in `bucket[hash_index]` from `NULL` to the address of the new data element by line 18. After the `while` loop, `ptrUpdate` points to the pointer that must be updated for the new element to be inserted into the appropriate bucket.

After the execution exits the `while` loop, lines 17 and 18 finish the job of inserting the new data element into the linked list. In the case where the bucket is empty, `ptrUpdate` will point to `bucket[hash_index]`, which contains a `NULL` pointer. Line 17 will then assign that `NULL` pointer to the next pointer of the new data element. Line 18 changes `bucket[hash_table]` to point to the new data element. In the case where the new data element is smaller than all elements in the linked list, `ptrUpdate` will also point to `bucket[hash_table]`, which points to the first element of the linked list. In this case, line 17 assigns the pointer to the first element of the linked list to the next pointer of the new data structure.

In the case where the new data element is greater than some of the linked list elements but smaller than the others, `ptrUpdate` will point to the `next` pointer of the element after which the new data element will be inserted. In this case, line 17 makes the new data element to point to the element right after the insertion point. Line 18 makes the original data element right before the insertion point to point to the new data element. The reader should verify that the code works correctly when the new data element is to be inserted to the end of the linked list.

Now that we have a good understanding of the C code, we will proceed with analyzing the amount of instruction level parallelism available in this piece of code.

3.1 [25/15/10/15/20/20/15] <2.1, 2.2, 3.2, 3.3, App. H> This part of our case study will focus on the amount of instruction level parallelism available to the *run time hardware scheduler* under the most favorable execution scenarios (the ideal case). (Later, we will consider less ideal scenarios for the run time hardware scheduler as well as the amount of parallelism available to a compiler scheduler.) For the ideal scenario, assume that the hash table is initially empty. Suppose there are 1024 new data elements, whose values are numbered sequentially from 0 to 1023, so that each goes in its own bucket (this reduces the problem to a matter of updating known array locations!). Figure 3.15 shows the hash table contents after the first three elements have been inserted, according to this "ideal case." Since the value of `element[i]` is simply `i` in this ideal case, each element is inserted into its own bucket.

For the purposes of this case study, assume that each line of code in Figure 3.14 takes one execution cycle (its dependence height is 1) and, for the purposes of computing ILP, takes one instruction. These (unrealistic) assumptions are made

to greatly simplify bookkeeping in solving the following exercises. Note that the `for` and `while` statements execute on each iteration of their respective loops, to test if the loop should continue. In this ideal case, most of the dependences in the code sequence are relaxed and a high degree of ILP is therefore readily available. We will later examine a general case, in which the realistic dependences in the code segment reduce the amount of parallelism available.

Further suppose that the code is executed on an "ideal" processor with infinite issue width, unlimited renaming, "omniscient" knowledge of memory access disambiguation, branch prediction, and so on, so that the execution of instructions is limited only by data dependence. Consider the following in this context:

a. [25] <2.1> Describe the data (true, anti, and output) and control dependences that govern the parallelism of this code segment, as seen by a run time hardware scheduler. Indicate only the *actual* dependences (i.e., ignore dependences between stores and loads that access different addresses, even if a compiler or processor would not realistically determine this). Draw the *dynamic* dependence graph for six consecutive iterations of the outer loop (for insertion of six elements), under the ideal case. Note that in this dynamic dependence graph, we are identifying data dependences between dynamic instances of instructions: each static instruction in the original program has multiple dynamic instances due to loop execution. *Hint:* The following definitions may help you find the dependences related to each instruction:

   - *Data true dependence:* On the results of which previous instructions does each instruction immediately depend?
   - *Data antidependence:* Which instructions subsequently write locations read by the instruction?
   - *Data output dependence:* Which instructions subsequently write locations written by the instruction?
   - *Control dependence:* On what previous decisions does the execution of a particular instruction depend (in what case will it be reached)?

b. [15] <2.1> Assuming the ideal case just described, and using the dynamic dependence graph you just constructed, how many instructions are executed, and in how many cycles?

c. [10] <3.2> What is the average level of ILP available during the execution of the `for` loop?

d. [15] <2.2, App. H> In part (c) we considered the maximum parallelism achievable by a run-time hardware scheduler using the code as written. How could a compiler increase the available parallelism, assuming that the compiler knows that it is dealing with the ideal case. *Hint:* Think about what is the primary constraint that prevents executing more iterations at once in the ideal case. How can the loop be restructured to relax that constraint?

e. [25] <3.2, 3.3> For simplicity, assume that only variables `i`, `hash_index`, `ptrCurr`, and `ptrUpdate` need to occupy registers. Assuming general renaming, how many registers are necessary to achieve the maximum achievable parallelism in part (b)?

f. [25] <3.3> Assume that in your answer to part (a) there are 7 instructions in each iteration. Now, assuming a consistent steady-state schedule of the instructions in the example and an issue rate of 3 instructions per cycle, how is execution time affected?

g. [15] <3.3> Finally, calculate the minimal instruction window size needed to achieve the maximal level of parallelism.

3.2 [15/15/15/10/10/15/15/10/10/10/25] <2.1, 3.2, 3.3> Let us now consider less favorable scenarios for extraction of instruction-level parallelism by a run-time hardware scheduler in the hash table code in Figure 3.14 (the general case). Suppose that there is no longer a guarantee that each bucket will receive exactly one item. Let us reevaluate our assessment of the parallelism available, given the more realistic situation, which adds some additional, important dependences.

Recall that in the ideal case, the relatively serial inner loop was not in play, and the outer loop provided ample parallelism. In general, the inner loop is in play: the inner `while` loop could iterate one or more times. Keep in mind that the inner loop, the `while` loop, has only a limited amount of instruction-level parallelism. First of all, each iteration of the `while` loop depends on the result of the previous iteration. Second, within each iteration, only a small number of instructions are executed.

The outer loop is, on the contrary, quite parallel. As long as two elements of the outer loop are hashed into different buckets, they can be entered in parallel. Even when they are hashed to the same bucket, they can still go in parallel as long as some type of memory disambiguation enforces correctness of memory loads and stores performed on behalf of each element.

Note that in reality, the data element values will likely be randomly distributed. Although we aim to provide the reader insight into more realistic execution scenarios, we will begin with some regular but nonideal data value patterns that are amenable to systematic analysis. These value patterns offer some intermediate steps toward understanding the amount of instruction-level parallelism under the most general, random data values.

a. [15] <2.1> Draw a dynamic dependence graph for the hash table code in Figure 3.14 when the values of the 1024 data elements to be inserted are 0, 1, 1024, 1025, 2048, 2049, 3072, 3073, ... Describe the new dependences across iterations for the `for` loop when the `while` loop is iterated one or more times. Pay special attention to the fact that the inner `while` loop now can iterate one or more times. The number of instructions in the outer `for` loop will therefore likely vary as it iterates. For the purpose of determining dependences between loads and stores, assume a dynamic memory disambiguation that cannot resolve the dependences between two memory

accesses based on different base pointer registers. For example, the run time hardware cannot disambiguate between a store based on `ptrUpdate` and a load based on `ptrCurr`.

b. [15] <2.1> Assuming the dynamic dependence graph you derived in part (a), how many instructions will be executed?

c. [15] <2.1> Assuming the dynamic dependence graph you derived in part (a) and an unlimited amount of hardware resources, how many clock cycles will it take to execute all the instructions you calculated in part (b)?

d. [10] <2.1> How much instruction-level parallelism is available in the dynamic dependence graph you derived in part (a)?

e. [10] <2.1, 3.2> Using the same assumption of run time memory disambiguation mechanism as in part (a), identify a sequence of data elements that will cause the worst-case scenario of the way these new dependences affect the level of parallelism available.

f. [15] <2.1, 3.2> Now, assume the worst-case sequence used in part (e), explain the potential effect of a perfect run time memory disambiguation mechanism (i.e., a system that tracks all outstanding stores and allows all nonconflicting loads to proceed). Derive the number of clock cycles required to execute all the instructions in the dynamic dependence graph.

   On the basis of what you have learned so far, consider a couple of qualitative questions: What is the effect of allowing loads to issue speculatively, before prior store addresses are known? How does such speculation affect the significance of memory latency in this code?

g. [15] <2.1, 3.2> Continue the same assumptions as in part (f), and calculate the number of instructions executed.

h. [10] <2.1, 3.2> Continue the same assumptions as in part (f), and calculate the amount of instruction-level parallelism available to the run-time hardware.

i. [10] <2.1, 3.2> In part (h), what is the effect of limited instruction window sizes on the level of instruction-level parallelism?

j. [10] <3.2, 3.3> Now, continuing to consider your solution to part (h), describe the cause of branch-prediction misses and the effect of each branch prediction on the level of parallelism available. Reflect briefly on the implications for power and efficiency. What are potential costs and benefits to executing many off-path speculative instructions (i.e., initiating execution of instructions that will later be squashed by branch-misprediction detection)? *Hint:* Think about the effect on the execution of subsequent insertions of mispredicting the number of elements before the insertion point.

k. [25] <3> Consider the concept of a static dependence graph that captures all the worst-case dependences for the purpose of constraining compiler scheduling and optimization. Draw the static dependence graph for the hash table code shown in Figure 3.14.

Compare the static dependence graph with the various dynamic dependence graphs drawn previously. Reflect in a paragraph or two on the implications of this comparison for dynamic and static discovery of instruction-level parallelism in this example's hash table code. In particular, how is the compiler constrained by having to consistently take into consideration the worst case, where a hardware mechanism might be free to take advantage opportunistically of fortuitous cases? What sort of approaches might help the compiler to make better use of this code?