G.1 Introduction: Exploiting Instruction-Level Parallelism Statically G-2
G.2 Detecting and Enhancing Loop-Level Parallelism G-2
G.3 Scheduling and Structuring Code for Parallelism G-12
G.4 Hardware Support for Exposing Parallelism: Predicated Instructions G-23
G.5 Hardware Support for Compiler Speculation G-27
G.6 The Intel IA-64 Architecture and Itanium Processor G-32
G.7 Concluding Remarks G-44

G

Hardware and Software for VLIW and EPIC

The EPIC approach is based on the application of massive resources. These resources include more load-store, computational, and branch units, as well as larger, lower-latency caches than would be required for a superscalar processor. Thus, IA-64 gambles that, in the future, power will not be the critical limitation, and that massive resources, along with the machinery to exploit them, will not penalize performance with their adverse effect on clock speed, path length, or CPI factors.

M. Hopkins [2000]
in a commentary on the EPIC approach and the IA-64 architecture

G.1 Introduction: Exploiting Instruction-Level Parallelism Statically

In this chapter we discuss compiler technology for increasing the amount of parallelism that we can exploit in a program as well as hardware support for these compiler techniques. The next section defines when a loop is parallel, how a dependence can prevent a loop from being parallel, and techniques for eliminating some types of dependences. The following section discusses the topic of scheduling code to improve parallelism. These two sections serve as an introduction to these techniques.

We do not attempt to explain the details of ILP-oriented compiler techniques, since that would take hundreds of pages, rather than the 20 we have allotted. Instead, we view this material as providing general background that will enable the reader to have a basic understanding of the compiler techniques used to exploit ILP in modern computers.

Hardware support for these compiler techniques can greatly increase their effectiveness, and Sections G.4 and G.5 explore such support. The IA-64 represents the culmination of the compiler and hardware ideas for exploiting parallelism statically and includes support for many of the concepts proposed by researchers during more than a decade of research into the area of compiler-based instruction-level parallelism. Section G.6 is a description and performance analyses of the Intel IA-64 architecture and its second-generation implementation, Itanium 2.

The core concepts that we exploit in statically based techniques—finding parallelism, reducing control and data dependences, and using speculation—are the same techniques we saw exploited in Chapter 2 using dynamic techniques. The key difference is that the techniques in this appendix are applied at compile time by the compiler, rather than at run time by the hardware. The advantages of compile time techniques are primarily two: they do not burden run time execution with any inefficiency, and they can take into account a wider range of the program than a run time approach might be able to incorporate. As an example of the latter, the next section shows how a compiler might determine that an entire loop can be executed in parallel; hardware techniques might or might not be able to find such parallelism. The major disadvantage of static approaches is that they can use only compile time information. Without run time information, compile time techniques must often be conservative and assume the worst case.

G.2 Detecting and Enhancing Loop-Level Parallelism

Loop-level parallelism is normally analyzed at the source level or close to it, while most analysis of ILP is done once instructions have been generated by the compiler. Loop-level analysis involves determining what dependences exist among the operands in a loop across the iterations of that loop. For now, we will

consider only data dependences, which arise when an operand is written at some point and read at a later point. Name dependences also exist and may be removed by renaming techniques like those we explored in Chapter 2.

The analysis of loop-level parallelism focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations; such a dependence is called a *loop-carried dependence*. Most of the examples we considered in Section 2.2 have no loop-carried dependences and, thus, are loop-level parallel. To see that a loop is parallel, let us first look at the source representation:

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

In this loop, there is a dependence between the two uses of `x[i]`, but this dependence is within a single iteration and is not loop carried. There is a dependence between successive uses of `i` in different iterations, which is loop carried, but this dependence involves an induction variable and can be easily recognized and eliminated. We saw examples of how to eliminate dependences involving induction variables during loop unrolling in Section 2.2, and we will look at additional examples later in this section.

Because finding loop-level parallelism involves recognizing structures such as loops, array references, and induction variable computations, the compiler can do this analysis more easily at or near the source level, as opposed to the machine-code level. Let's look at a more complex example.

Example Consider a loop like this one:

```
for (i=1; i<=100; i=i+1) {
      A[i+1] = A[i] + C[i];   /* S1 */
      B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

Assume that `A`, `B`, and `C` are distinct, nonoverlapping arrays. (In practice, the arrays may sometimes be the same or may overlap. Because the arrays may be passed as parameters to a procedure, which includes this loop, determining whether arrays overlap or are identical often requires sophisticated, interprocedural analysis of the program.) What are the data dependences among the statements S1 and S2 in the loop?

Answer There are two different dependences:

1. S1 uses a value computed by S1 in an earlier iteration, since iteration `i` computes `A[i+1]`, which is read in iteration `i+1`. The same is true of S2 for `B[i]` and `B[i+1]`.
2. S2 uses the value, `A[i+1]`, computed by S1 in the same iteration.

These two dependences are different and have different effects. To see how they differ, let's assume that only one of these dependences exists at a time. Because the dependence of statement S1 is on an earlier iteration of S1, this dependence is loop carried. This dependence forces successive iterations of this loop to execute in series.

The second dependence (S2 depending on S1) is within an iteration and is not loop carried. Thus, if this were the only dependence, multiple iterations of the loop could execute in parallel, as long as each pair of statements in an iteration were kept in order. We saw this type of dependence in an example in Section 2.2, where unrolling was able to expose the parallelism.

It is also possible to have a loop-carried dependence that does not prevent parallelism, as the next example shows.

Example Consider a loop like this one:

```
for (i=1; i<=100; i=i+1) {
      A[i] = A[i] + B[i];   /* S1 */
      B[i+1] = C[i] + D[i]; /* S2 */
}
```

What are the dependences between S1 and S2? Is this loop parallel? If not, show how to make it parallel.

Answer Statement S1 uses the value assigned in the previous iteration by statement S2, so there is a loop-carried dependence between S2 and S1. Despite this loop-carried dependence, this loop can be made parallel. Unlike the earlier loop, this dependence is not circular: neither statement depends on itself, and although S1 depends on S2, S2 does not depend on S1. A loop is parallel if it can be written without a cycle in the dependences, since the absence of a cycle means that the dependences give a partial ordering on the statements.

Although there are no circular dependences in the above loop, it must be transformed to conform to the partial ordering and expose the parallelism. Two observations are critical to this transformation:

1. There is no dependence from S1 to S2. If there were, then there would be a cycle in the dependences and the loop would not be parallel. Since this other dependence is absent, interchanging the two statements will not affect the execution of S2.
2. On the first iteration of the loop, statement S1 depends on the value of `B[1]` computed prior to initiating the loop.

These two observations allow us to replace the loop above with the following code sequence:

```
A[1] = A[1] + B[1];
for (i=1; i<=99; i=i+1) {
      B[i+1] = C[i] + D[i];
      A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
```

The dependence between the two statements is no longer loop carried, so that iterations of the loop may be overlapped, provided the statements in each iteration are kept in order.

Our analysis needs to begin by finding all loop-carried dependences. This dependence information is *inexact,* in the sense that it tells us that such a dependence *may* exist. Consider the following example:

```
for (i=1;i<=100;i=i+1) {
      A[i] = B[i] + C[i]
      D[i] = A[i] * E[i]
}
```

The second reference to A in this example need not be translated to a load instruction, since we know that the value is computed and stored by the previous statement; hence, the second reference to A can simply be a reference to the register into which A was computed. Performing this optimization requires knowing that the two references are *always* to the same memory address and that there is no intervening access to the same location. Normally, data dependence analysis only tells that one reference *may* depend on another; a more complex analysis is required to determine that two references *must be* to the exact same address. In the example above, a simple version of this analysis suffices, since the two references are in the same basic block.

Often loop-carried dependences are in the form of a *recurrence:*

```
for (i=2;i<=100;i=i+1) {
      Y[i] = Y[i-1] + Y[i];
}
```

A recurrence is when a variable is defined based on the value of that variable in an earlier iteration, often the one immediately preceding, as in the above fragment. Detecting a recurrence can be important for two reasons: Some architectures (especially vector computers) have special support for executing recurrences, and some recurrences can be the source of a reasonable amount of parallelism. To see how the latter can be true, consider this loop:

```
for (i=6;i<=100;i=i+1) {
      Y[i] = Y[i-5] + Y[i];
}
```

On the iteration *i,* the loop references element $i - 5$. The loop is said to have a *dependence distance* of 5. Many loops with carried dependences have a dependence distance of 1. The larger the distance, the more potential parallelism can be obtained by unrolling the loop. For example, if we unroll the first loop, with a dependence distance of 1, successive statements are dependent on one another; there is still some parallelism among the individual instructions, but not much. If we unroll the loop that has a dependence distance of 5, there is a sequence of five statements that have no dependences, and thus much more ILP. Although many loops with loop-carried dependences have a dependence distance of 1, cases with larger distances do arise, and the longer distance may well provide enough parallelism to keep a processor busy.

Finding Dependences

Finding the dependences in a program is an important part of three tasks: (1) good scheduling of code, (2) determining which loops might contain parallelism, and (3) eliminating name dependences. The complexity of dependence analysis arises because of the presence of arrays and pointers in languages like C or C++, or pass-by-reference parameter passing in FORTRAN. Since scalar variable references explicitly refer to a name, they can usually be analyzed quite easily, with aliasing because of pointers and reference parameters causing some complications and uncertainty in the analysis.

How does the compiler detect dependences in general? Nearly all dependence analysis algorithms work on the assumption that array indices are *affine.* In simplest terms, a one-dimensional array index is affine if it can be written in the form $a \times i + b$, where a and b are constants and i is the loop index variable. The index of a multidimensional array is affine if the index in each dimension is affine. Sparse array accesses, which typically have the form `x[y[i]]`, are one of the major examples of nonaffine accesses.

Determining whether there is a dependence between two references to the same array in a loop is thus equivalent to determining whether two affine functions can have the same value for different indices between the bounds of the loop. For example, suppose we have stored to an array element with index value $a \times i + b$ and loaded from the same array with index value $c \times i + d$, where i is the for-loop index variable that runs from m to n. A dependence exists if two conditions hold:

1. There are two iteration indices, j and k, both within the limits of the for loop. That is, $m \leq j \leq n$, $m \leq k \leq n$.
2. The loop stores into an array element indexed by $a \times j + b$ and later fetches from that *same* array element when it is indexed by $c \times k + d$. That is, $a \times j + b = c \times k + d$.

In general, we cannot determine whether a dependence exists at compile time. For example, the values of *a, b, c,* and *d* may not be known (they could be values in other arrays), making it impossible to tell if a dependence exists. In other cases, the dependence testing may be very expensive but decidable at compile time. For example, the accesses may depend on the iteration indices of multiple nested loops. Many programs, however, contain primarily simple indices where *a, b, c,* and *d* are all constants. For these cases, it is possible to devise reasonable compile time tests for dependence.

As an example, a simple and sufficient test for the absence of a dependence is the *greatest common divisor* (GCD) test. It is based on the observation that if a loop-carried dependence exists, then GCD (c,a) must divide $(d - b)$. (Recall that an integer, *x, divides* another integer, *y,* if we get an integer quotient when we do the division y/x and there is no remainder.)

Example Use the GCD test to determine whether dependences exist in the following loop:

```
for (i=1; i<=100; i=i+1) {
      X[2*i+3] = X[2*i] * 5.0;
}
```

Answer Given the values $a = 2$, $b = 3$, $c = 2$, and $d = 0$, then GCD$(a,c) = 2$, and $d - b = -3$. Since 2 does not divide –3, no dependence is possible.

The GCD test is sufficient to guarantee that no dependence exists; however, there are cases where the GCD test succeeds but no dependence exists. This can arise, for example, because the GCD test does not take the loop bounds into account.

In general, determining whether a dependence actually exists is NP-complete. In practice, however, many common cases can be analyzed precisely at low cost. Recently, approaches using a hierarchy of exact tests increasing in generality and cost have been shown to be both accurate and efficient. (A test is *exact* if it precisely determines whether a dependence exists. Although the general case is NP-complete, there exist exact tests for restricted situations that are much cheaper.)

In addition to detecting the presence of a dependence, a compiler wants to classify the type of dependence. This classification allows a compiler to recognize name dependences and eliminate them at compile time by renaming and copying.

Example The following loop has multiple types of dependences. Find all the true dependences, output dependences, and antidependences, and eliminate the output dependences and antidependences by renaming.

```
for (i=1; i<=100; i=i+1) {
    Y[i] = X[i] / c; /* S1 */
    X[i] = X[i] + c; /* S2 */
    Z[i] = Y[i] + c; /* S3 */
    Y[i] = c - Y[i]; /* S4 */
}
```

Answer The following dependences exist among the four statements:

1. There are true dependences from S1 to S3 and from S1 to S4 because of `Y[i]`. These are not loop carried, so they do not prevent the loop from being considered parallel. These dependences will force S3 and S4 to wait for S1 to complete.
2. There is an antidependence from S1 to S2, based on `X[i]`.
3. There is an antidependence from S3 to S4 for `Y[i]`.
4. There is an output dependence from S1 to S4, based on `Y[i]`.

The following version of the loop eliminates these false (or pseudo) dependences.

```
for (i=1; i<=100; i=i+1 {
    /* Y renamed to T to remove output dependence */
    T[i] = X[i] / c;
    /* X renamed to X1 to remove antidependence */
    X1[i] = X[i] + c;
    /* Y renamed to T to remove antidependence */
    Z[i] = T[i] + c;
    Y[i] = c - T[i];
}
```

After the loop, the variable `X` has been renamed `X1`. In code that follows the loop, the compiler can simply replace the name `X` by `X1`. In this case, renaming does not require an actual copy operation but can be done by substituting names or by register allocation. In other cases, however, renaming will require copying.

Dependence analysis is a critical technology for exploiting parallelism. At the instruction level it provides information needed to interchange memory references when scheduling, as well as to determine the benefits of unrolling a loop. For detecting loop-level parallelism, dependence analysis is the basic tool. Effectively compiling programs to either vector computers or multiprocessors depends critically on this analysis. The major drawback of dependence analysis is that it applies only under a limited set of circumstances, namely, among references within a single loop nest and using affine index functions. Thus, there is a wide variety of situations in which array-oriented dependence analysis *cannot* tell us what we might want to know, including the following:

- When objects are referenced via pointers rather than array indices (but see discussion below)
- When array indexing is indirect through another array, which happens with many representations of sparse arrays
- When a dependence may exist for some value of the inputs, but does not exist in actuality when the code is run since the inputs never take on those values
- When an optimization depends on knowing more than just the possibility of a dependence, but needs to know on *which* write of a variable does a read of that variable depend

To deal with the issue of analyzing programs with pointers, another type of analysis, often called *points-to* analysis, is required (see Wilson and Lam [1995]). The key question that we want answered from dependence analysis of pointers is whether two pointers can designate the same address. In the case of complex dynamic data structures, this problem is extremely difficult. For example, we may want to know whether two pointers can reference the *same* node in a list at a given point in a program, which in general is undecidable and in practice is extremely difficult to answer. We may, however, be able to answer a simpler question: Can two pointers designate nodes in the *same* list, even if they may be separate nodes? This more restricted analysis can still be quite useful in scheduling memory accesses performed through pointers.

The basic approach used in points-to analysis relies on information from three major sources:

1. Type information, which restricts what a pointer can point to.
2. Information derived when an object is allocated or when the address of an object is taken, which can be used to restrict what a pointer can point to. For example, if p always points to an object allocated in a given source line and q never points to that object, then p and q can never point to the same object.
3. Information derived from pointer assignments. For example, if p may be assigned the value of q, then p may point to anything q points to.

There are several cases where analyzing pointers has been successfully applied and is extremely useful:

- When pointers are used to pass the address of an object as a parameter, it is possible to use points-to analysis to determine the possible set of objects referenced by a pointer. One important use is to determine if two pointer parameters may designate the same object.
- When a pointer can point to one of several types, it is sometimes possible to determine the type of the data object that a pointer designates at different parts of the program.
- It is often possible to separate out pointers that may only point to a local object versus a global one.

There are two different types of limitations that affect our ability to do accurate dependence analysis for large programs. The first type of limitation arises from restrictions in the analysis algorithms. Often, we are limited by the lack of applicability of the analysis rather than a shortcoming in dependence analysis per se. For example, dependence analysis for pointers is essentially impossible for programs that use pointers in arbitrary fashion—for example, by doing arithmetic on pointers.

The second limitation is the need to analyze behavior across procedure boundaries to get accurate information. For example, if a procedure accepts two parameters that are pointers, determining whether the values could be the same requires analyzing across procedure boundaries. This type of analysis, called *interprocedural analysis,* is much more difficult and complex than analysis within a single procedure. Unlike the case of analyzing array indices within a single loop nest, points-to analysis usually requires an interprocedural analysis. The reason for this is simple. Suppose we are analyzing a program segment with two pointers; if the analysis does not know anything about the two pointers at the start of the program segment, it must be conservative and assume the worst case. The worst case is that the two pointers *may* designate the same object, but they are not *guaranteed* to designate the same object. This worst case is likely to propagate through the analysis, producing useless information. In practice, getting fully accurate interprocedural information is usually too expensive for real programs. Instead, compilers usually use approximations in interprocedural analysis. The result is that the information may be too inaccurate to be useful.

Modern programming languages that use strong typing, such as Java, make the analysis of dependences easier. At the same time the extensive use of procedures to structure programs, as well as abstract data types, makes the analysis more difficult. Nonetheless, we expect that continued advances in analysis algorithms, combined with the increasing importance of pointer dependency analysis, will mean that there is continued progress on this important problem.

Eliminating Dependent Computations

Compilers can reduce the impact of dependent computations so as to achieve more ILP. The key technique is to eliminate or reduce a dependent computation by back substitution, which increases the amount of parallelism and sometimes increases the amount of computation required. These techniques can be applied both within a basic block and within loops, and we describe them differently.

Within a basic block, algebraic simplifications of expressions and an optimization called *copy propagation,* which eliminates operations that copy values, can be used to simplify sequences like the following:

```
DADDUI    R1,R2,#4
DADDUI    R1,R1,#4
```

to

```
DADDUI      R1,R2,#8
```

assuming this is the only use of R1. In fact, the techniques we used to reduce multiple increments of array indices during loop unrolling and to move the increments across memory addresses in Section 2.2 are examples of this type of optimization.

In these examples, computations are actually eliminated, but it is also possible that we may want to increase the parallelism of the code, possibly even increasing the number of operations. Such optimizations are called *tree height reduction,* since they reduce the height of the tree structure representing a computation, making it wider but shorter. Consider the following code sequence:

```
ADD         R1,R2,R3
ADD         R4,R1,R6
ADD         R8,R4,R7
```

Notice that this sequence requires at least three execution cycles, since all the instructions depend on the immediate predecessor. By taking advantage of associativity, we can transform the code and rewrite it as

```
ADD         R1,R2,R3
ADD         R4,R6,R7
ADD         R8,R1,R4
```

This sequence can be computed in two execution cycles. When loop unrolling is used, opportunities for these types of optimizations occur frequently.

Although arithmetic with unlimited range and precision is associative, computer arithmetic is not associative, for either integer arithmetic, because of limited range, or floating-point arithmetic, because of both range and precision. Thus, using these restructuring techniques can sometimes lead to erroneous behavior, although such occurrences are rare. For this reason, most compilers require that optimizations that rely on associativity be explicitly enabled.

When loops are unrolled, this sort of algebraic optimization is important to reduce the impact of dependences arising from recurrences. *Recurrences* are expressions whose value on one iteration is given by a function that depends on the previous iterations. When a loop with a recurrence is unrolled, we may be able to algebraically optimize the unrolled loop, so that the recurrence need only be evaluated once per unrolled iteration. One common type of recurrence arises from an explicit program statement, such as

```
sum = sum + x;
```

Assume we unroll a loop with this recurrence five times. If we let the value of x on these five iterations be given by x1, x2, x3, x4, and x5, then we can write the value of sum at the end of each unroll as

```
sum = sum + x1 + x2 + x3 + x4 + x5;
```

If unoptimized this expression requires five dependent operations, but it can be rewritten as

```
sum = ((sum + x1) + (x2 + x3)) + (x4 + x5);
```

which can be evaluated in only three dependent operations.

Recurrences also arise from implicit calculations, such as those associated with array indexing. Each array index translates to an address that is computed based on the loop index variable. Again, with unrolling and algebraic optimization, the dependent computations can be minimized.

G.3 Scheduling and Structuring Code for Parallelism

We have already seen that one compiler technique, loop unrolling, is useful to uncover parallelism among instructions by creating longer sequences of straight-line code. There are two other important techniques that have been developed for this purpose: *software pipelining* and *trace scheduling.*

Software Pipelining: Symbolic Loop Unrolling

Software pipelining is a technique for reorganizing loops such that each iteration in the software-pipelined code is made from instructions chosen from different iterations of the original loop. This approach is most easily understood by looking at the scheduled code for the unrolled loop, which appeared in the example on page 78. The scheduler in this example essentially interleaves instructions from different loop iterations, so as to separate the dependent instructions that occur within a single loop iteration. By choosing instructions from different iterations, dependent computations are separated from one another by an entire loop body, increasing the possibility that the unrolled loop can be scheduled without stalls.

A software-pipelined loop interleaves instructions from different iterations without unrolling the loop, as illustrated in Figure G.1. This technique is the software counterpart to what Tomasulo's algorithm does in hardware. The software-pipelined loop for the earlier example would contain one load, one add, and one store, each from a different iteration. There is also some start-up code that is needed before the loop begins as well as code to finish up after the loop is completed. We will ignore these in this discussion, for simplicity.

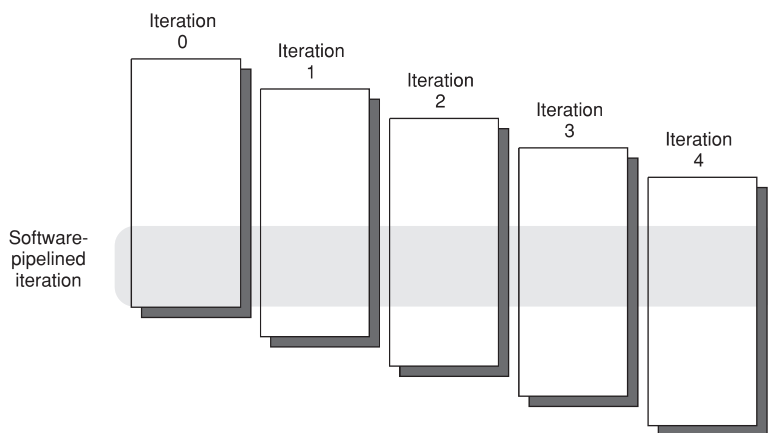

Figure G.1 A software-pipelined loop chooses instructions from different loop iterations, thus separating the dependent instructions within one iteration of the original loop. The start-up and finish-up code will correspond to the portions above and below the software-pipelined iteration.

Example Show a software-pipelined version of this loop, which increments all the elements of an array whose starting address is in R1 by the contents of F2:

```
Loop:   L.D      F0,0(R1)
        ADD.D    F4,F0,F2
        S.D      F4,0(R1)
        DADDUI   R1,R1,#-8
        BNE      R1,R2,Loop
```

You may omit the start-up and clean-up code.

Answer Software pipelining symbolically unrolls the loop and then selects instructions from each iteration. Since the unrolling is symbolic, the loop overhead instructions (the DADDUI and BNE) need not be replicated. Here's the body of the unrolled loop without overhead instructions, highlighting the instructions taken from each iteration:

```
Iteration i:    L.D     F0,0(R1)
                ADD.D   F4,F0,F2
                S.D     F4,0(R1)
Iteration i+1:  L.D     F0,0(R1)
                ADD.D   F4,F0,F2
                S.D     F4,0(R1)
Iteration i+2:  L.D     F0,0(R1)
                ADD.D   F4,F0,F2
                S.D     F4,0(R1)
```

The selected instructions from different iterations are then put together in the loop with the loop control instructions:

```
Loop:   S.D      F4,16(R1)     ;stores into M[i]
        ADD.D    F4,F0,F2      ;adds to M[i-1]
        L.D      F0,0(R1)      ;loads M[i-2]
        DADDUI   R1,R1,#-8
        BNE      R1,R2,Loop
```

This loop can be run at a rate of 5 cycles per result, ignoring the start-up and clean-up portions, and assuming that DADDUI is scheduled before the ADD.D and that the L.D instruction, with an adjusted offset, is placed in the branch delay slot. Because the load and store are separated by offsets of 16 (two iterations), the loop should run for two fewer iterations. Notice that the reuse of registers (e.g., F4, F0, and R1) requires the hardware to avoid the WAR hazards that would occur in the loop. This hazard should not be a problem in this case, since no data-dependent stalls should occur.

By looking at the unrolled version we can see what the start-up code and finish-up code will need to be. For start-up, we will need to execute any instructions that correspond to iteration 1 and 2 that will not be executed. These instructions are the L.D for iterations 1 and 2 and the ADD.D for iteration 1. For the finish-up code, we need to execute any instructions that will not be executed in the final two iterations. These include the ADD.D for the last iteration and the S.D for the last two iterations.

Register management in software-pipelined loops can be tricky. The previous example is not too hard since the registers that are written on one loop iteration are read on the next. In other cases, we may need to increase the number of iterations between when we issue an instruction and when the result is used. This increase is required when there are a small number of instructions in the loop body and the latencies are large. In such cases, a combination of software pipelining and loop unrolling is needed.

Software pipelining can be thought of as *symbolic* loop unrolling. Indeed, some of the algorithms for software pipelining use loop-unrolling algorithms to figure out how to software-pipeline the loop. The major advantage of software pipelining over straight loop unrolling is that software pipelining consumes less code space. Software pipelining and loop unrolling, in addition to yielding a better scheduled inner loop, each reduce a different type of overhead. Loop unrolling reduces the overhead of the loop—the branch and counter update code. Software pipelining reduces the time when the loop is not running at peak speed to once per loop at the beginning and end. If we unroll a loop that does 100 iterations a constant number of times, say, 4, we pay the overhead 100/4 = 25 times—every time the inner unrolled loop is initiated. Figure G.2 shows this behavior graphically. Because these techniques attack two different types of overhead, the best performance can come from doing both. In practice, compilation using soft-

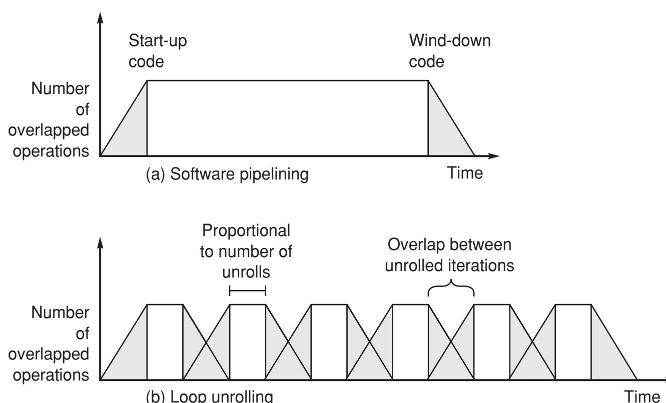

Figure G.2 The execution pattern for (a) a software-pipelined loop and (b) an unrolled loop. The shaded areas are the times when the loop is not running with maximum overlap or parallelism among instructions. This occurs once at the beginning and once at the end for the software-pipelined loop. For the unrolled loop it occurs *m/n* times if the loop has a total of *m* iterations and is unrolled *n* times. Each block represents an unroll of *n* iterations. Increasing the number of unrollings will reduce the start-up and clean-up overhead. The overhead of one iteration overlaps with the overhead of the next, thereby reducing the impact. The total area under the polygonal region in each case will be the same, since the total number of operations is just the execution rate multiplied by the time.

ware pipelining is quite difficult for several reasons: Many loops require significant transformation before they can be software pipelined, the trade-offs in terms of overhead versus efficiency of the software-pipelined loop are complex, and the issue of register management creates additional complexities. To help deal with the last two of these issues, the IA-64 added extensive hardware support for software pipelining. Although this hardware can make it more efficient to apply software pipelining, it does not eliminate the need for complex compiler support, or the need to make difficult decisions about the best way to compile a loop.

Global Code Scheduling

In Section 2.2 we examined the use of loop unrolling and code scheduling to improve ILP. The techniques in Section 2.2 work well when the loop body is straight-line code, since the resulting unrolled loop looks like a single basic block. Similarly, software pipelining works well when the body is a single basic block, since it is easier to find the repeatable schedule. When the body of an unrolled loop contains internal control flow, however, scheduling the code is much more complex. In general, effective scheduling of a loop body with internal control flow will require moving instructions across branches, which is global code scheduling. In this section, we first examine the challenge and limitations of

global code scheduling. In Section G.4 we examine hardware support for eliminating control flow within an inner loop; then, we examine two compiler techniques that can be used when eliminating the control flow is not a viable approach.

Global code scheduling aims to compact a code fragment with internal control structure into the shortest possible sequence that preserves the data and control dependences. The data dependences force a partial order on operations, while the control dependences dictate instructions across which code cannot be easily moved. Data dependences are overcome by unrolling and, in the case of memory operations, using dependence analysis to determine if two references refer to the same address. Finding the shortest possible sequence for a piece of code means finding the shortest sequence for the *critical path,* which is the longest sequence of dependent instructions.

Control dependences arising from loop branches are reduced by unrolling. Global code scheduling can reduce the effect of control dependences arising from conditional nonloop branches by moving code. Since moving code across branches will often affect the frequency of execution of such code, effectively using global code motion requires estimates of the relative frequency of different paths. Although global code motion cannot guarantee faster code, if the frequency information is accurate, the compiler can determine whether such code movement is likely to lead to faster code.

Global code motion is important since many inner loops contain conditional statements. Figure G.3 shows a typical code fragment, which may be thought of as an iteration of an unrolled loop, and highlights the more common control flow.

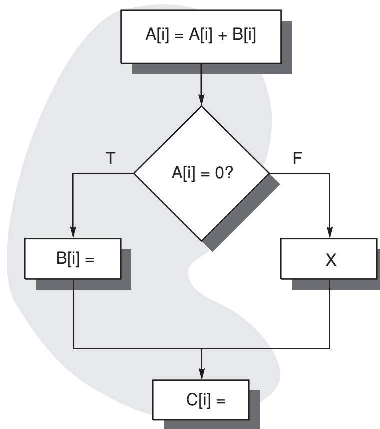

Figure G.3 A code fragment and the common path shaded with gray. Moving the assignments to B or C requires a more complex analysis than for straight-line code. In this section we focus on scheduling this code segment efficiently without hardware assistance. Predication or conditional instructions, which we discuss in the next section, provide another way to schedule this code.

Effectively scheduling this code could require that we move the assignments to B and C to earlier in the execution sequence, before the test of A. Such global code motion must satisfy a set of constraints to be legal. In addition, the movement of the code associated with B, unlike that associated with C, is speculative: It will speed the computation up only when the path containing the code would be taken.

To perform the movement of B, we must ensure that neither the data flow nor the exception behavior is changed. Compilers avoid changing the exception behavior by not moving certain classes of instructions, such as memory references, that can cause exceptions. In Section G.5, we will see how hardware support allows for more opportunities for speculative code motion as well as removes control dependences. Although such enhanced support for speculation can make it possible to explore more opportunities, the difficulty of choosing how to best compile the code remains complex.

How can the compiler ensure that the assignments to B and C can be moved without affecting the data flow? To see what's involved, let's look at a typical code generation sequence for the flowchart in Figure G.3. Assuming that the addresses for A, B, C are in R1, R2, and R3, respectively, here is such a sequence:

```
            LD      R4,0(R1)        ;load A
            LD      R5,0(R2)        ;load B
            DADDU   R4,R4,R5        ;Add to A
            SD      R4,0(R1)        ;Store A
            ...
            BNEZ    R4,elsepart     ;Test A
            ...                     ;then part
            SD      ...,0(R2)       ;Stores to B
            ...
            J       join            ;jump over else
elsepart:   ...                     ;else part
            X                       ;code for X
            ...
join:       ...                     ;after if
            SD      ...,0(R3)       ;store C[i]
```

Let's first consider the problem of moving the assignment to B to before the BNEZ instruction. Call the last instruction to assign to B before the if statement *i.* If B is referenced before it is assigned either in code segment X or after the if statement, call the referencing instruction *j.* If there is such an instruction *j,* then moving the assignment to B will change the data flow of the program. In particular, moving the assignment to B will cause *j* to become data dependent on the moved version of the assignment to B rather than on *i,* on which *j* originally depended. You could imagine more clever schemes to allow B to be moved even when the value is used: For example, in the first case, we could make a shadow copy of B before the if statement and use that shadow copy in X. Such schemes are usually avoided, both because they are complex to implement and because

they will slow down the program if the trace selected is not optimal and the operations end up requiring additional instructions to execute.

Moving the assignment to C up to before the first branch requires two steps. First, the assignment is moved over the join point of the else part into the portion corresponding to the then part. This movement makes the instructions for C control dependent on the branch and means that they will not execute if the else path, which is the infrequent path, is chosen. Hence, instructions that were data dependent on the assignment to C, and which execute after this code fragment, will be affected. To ensure the correct value is computed for such instructions, a copy is made of the instructions that compute and assign to C on the else path. Second, we can move C from the then part of the branch across the branch condition, if it does not affect any data flow into the branch condition. If C is moved to before the if test, the copy of C in the else branch can usually be eliminated, since it will be redundant.

We can see from this example that global code scheduling is subject to many constraints. This observation is what led designers to provide hardware support to make such code motion easier, and Sections G.4 and G.5 explores such support in detail.

Global code scheduling also requires complex trade-offs to make code motion decisions. For example, assuming that the assignment to B can be moved before the conditional branch (possibly with some compensation code on the alternative branch), will this movement make the code run faster? The answer is, possibly! Similarly, moving the copies of C into the if and else branches makes the code initially bigger! Only if the compiler can successfully move the computation across the if test will there be a likely benefit.

Consider the factors that the compiler would have to consider in moving the computation and assignment of B:

- What are the relative execution frequencies of the then case and the else case in the branch? If the then case is much more frequent, the code motion may be beneficial. If not, it is less likely, although not impossible, to consider moving the code.
- What is the cost of executing the computation and assignment to B above the branch? It may be that there are a number of empty instruction issue slots in the code above the branch and that the instructions for B can be placed into these slots that would otherwise go empty. This opportunity makes the computation of B "free" at least to first order.
- How will the movement of B change the execution time for the then case? If B is at the start of the critical path for the then case, moving it may be highly beneficial.
- Is B the best code fragment that can be moved above the branch? How does it compare with moving C or other statements within the then case?
- What is the cost of the compensation code that may be necessary for the else case? How effectively can this code be scheduled, and what is its impact on execution time?

As we can see from this *partial* list, global code scheduling is an extremely complex problem. The trade-offs depend on many factors, and individual decisions to globally schedule instructions are highly interdependent. Even choosing which instructions to start considering as candidates for global code motion is complex!

To try to simplify this process, several different methods for global code scheduling have been developed. The two methods we briefly explore here rely on a simple principle: focus the attention of the compiler on a straight-line code segment representing what is estimated to be the most frequently executed code path. Unrolling is used to generate the straight-line code, but, of course, the complexity arises in how conditional branches are handled. In both cases, they are effectively straightened by choosing and scheduling the most frequent path.

Trace Scheduling: Focusing on the Critical Path

Trace scheduling is useful for processors with a large number of issues per clock, where conditional or predicated execution (see Section G.4) is inappropriate or unsupported, and where simple loop unrolling may not be sufficient by itself to uncover enough ILP to keep the processor busy. Trace scheduling is a way to organize the global code motion process, so as to simplify the code scheduling by incurring the costs of possible code motion on the less frequent paths. Because it can generate *significant* overheads on the designated infrequent path, it is best used where profile information indicates significant differences in frequency between different paths and where the profile information is highly indicative of program behavior independent of the input. Of course, this limits its effective applicability to certain classes of programs.

There are two steps to trace scheduling. The first step, called *trace selection,* tries to find a likely sequence of basic blocks whose operations will be put together into a smaller number of instructions; this sequence is called a *trace*. Loop unrolling is used to generate long traces, since loop branches are taken with high probability. Additionally, by using static branch prediction, other conditional branches are also chosen as taken or not taken, so that the resultant trace is a straight-line sequence resulting from concatenating many basic blocks. If, for example, the program fragment shown in Figure G.3 corresponds to an inner loop with the highlighted path being much more frequent, and the loop were unwound four times, the primary trace would consist of four copies of the shaded portion of the program, as shown in Figure G.4.

Once a trace is selected, the second process, called *trace compaction,* tries to squeeze the trace into a small number of wide instructions. Trace compaction is code scheduling; hence, it attempts to move operations as early as it can in a sequence (trace), packing the operations into as few wide instructions (or issue packets) as possible.

The advantage of the trace scheduling approach is that it simplifies the decisions concerning global code motion. In particular, branches are viewed as jumps into or out of the selected trace, which is assumed to be the most probable path.

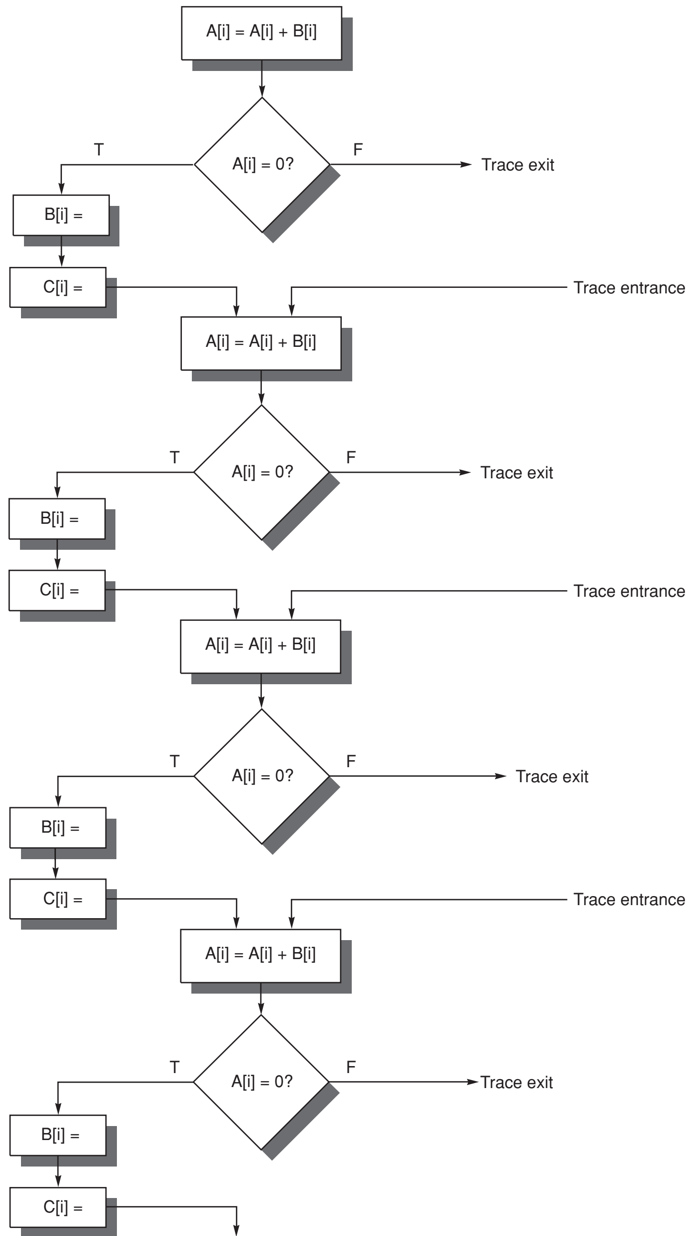

Figure G.4 This trace is obtained by assuming that the program fragment in Figure G.3 is the inner loop and unwinding it four times, treating the shaded portion in Figure G.3 as the likely path. The trace exits correspond to jumps off the frequent path, and the trace entrances correspond to returns to the trace.

When code is moved across such trace entry and exit points, additional bookkeeping code will often be needed on the entry or exit point. The key assumption is that the trace is so much more probable than the alternatives that the cost of the bookkeeping code need not be a deciding factor: If an instruction can be moved and thereby make the main trace execute faster, it is moved.

Although trace scheduling has been successfully applied to scientific code with its intensive loops and accurate profile data, it remains unclear whether this approach is suitable for programs that are less simply characterized and less loop-intensive. In such programs, the significant overheads of compensation code may make trace scheduling an unattractive approach, or, at best, its effective use will be extremely complex for the compiler.

Superblocks

One of the major drawbacks of trace scheduling is that the entries and exits into the middle of the trace cause significant complications, requiring the compiler to generate and track the compensation code and often making it difficult to assess the cost of such code. *Superblocks* are formed by a process similar to that used for traces, but are a form of extended basic blocks, which are restricted to a single entry point but allow multiple exits.

Because superblocks have only a single entry point, compacting a superblock is easier than compacting a trace since only code motion across an exit need be considered. In our earlier example, we would form superblocks that contained only one entrance and, hence, moving C would be easier. Furthermore, in loops that have a single loop exit based on a count (for example, a for loop with no loop exit other than the loop termination condition), the resulting superblocks have only one exit as well as one entrance. Such blocks can then be scheduled more easily.

How can a superblock with only one entrance be constructed? The answer is to use *tail duplication* to create a separate block that corresponds to the portion of the trace after the entry. In our previous example, each unrolling of the loop would create an exit from the superblock to a residual loop that handles the remaining iterations. Figure G.5 shows the superblock structure if the code fragment from Figure G.3 is treated as the body of an inner loop and unrolled four times. The residual loop handles any iterations that occur if the superblock is exited, which, in turn, occurs when the unpredicted path is selected. If the expected frequency of the residual loop were still high, a superblock could be created for that loop as well.

The superblock approach reduces the complexity of bookkeeping and scheduling versus the more general trace generation approach, but may enlarge code size more than a trace-based approach. Like trace scheduling, superblock scheduling may be most appropriate when other techniques (e.g., if conversion) fail. Even in such cases, assessing the cost of code duplication may limit the usefulness of the approach and will certainly complicate the compilation process.

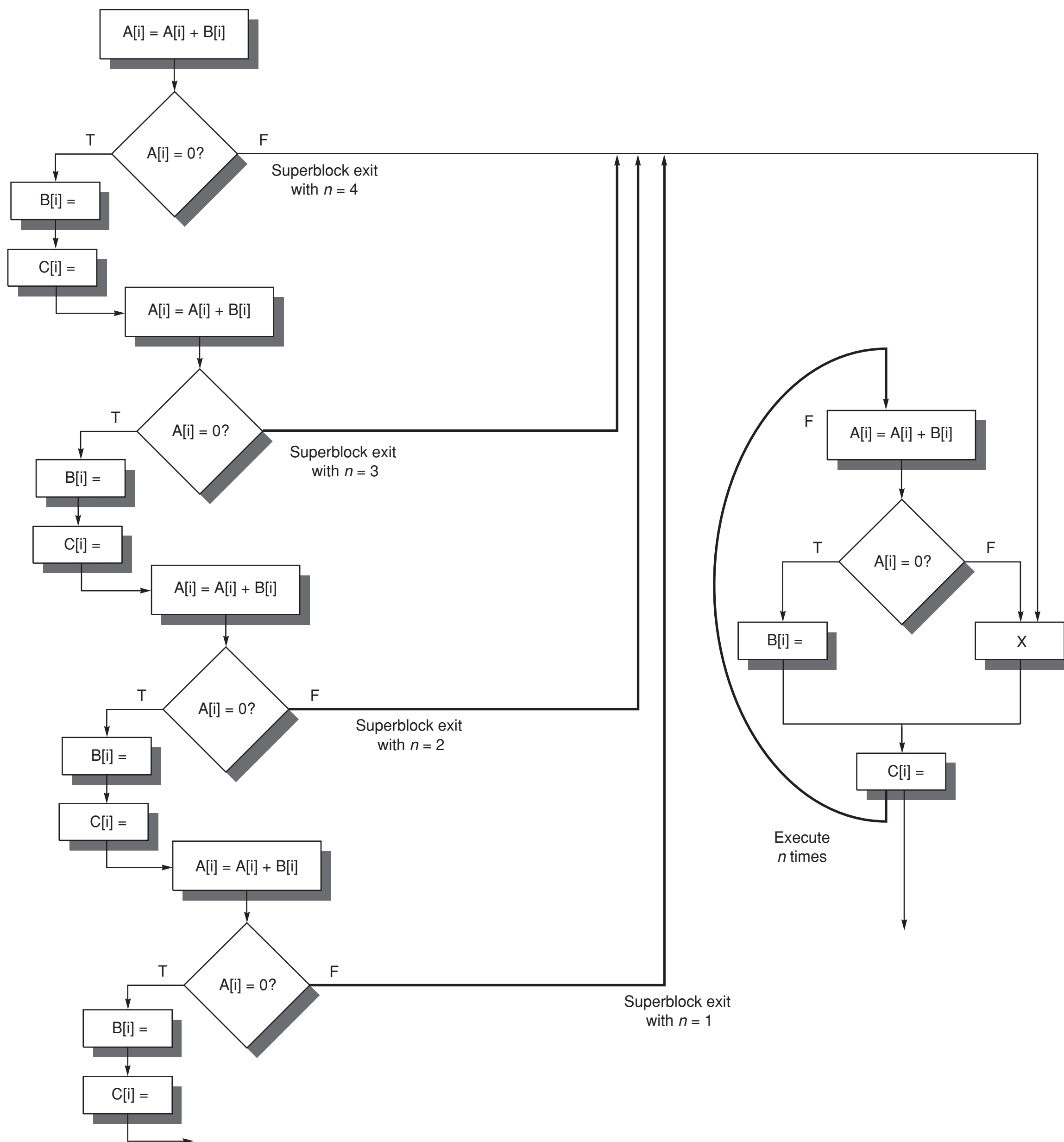

Figure G.5 This superblock results from unrolling the code in Figure G.3 four times and creating a superblock.

Loop unrolling, software pipelining, trace scheduling, and superblock scheduling all aim at trying to increase the amount of ILP that can be exploited by a processor issuing more than one instruction on every clock cycle. The effective-

ness of each of these techniques and their suitability for various architectural approaches are among the hottest topics being actively pursued by researchers and designers of high-speed processors.

G.4 Hardware Support for Exposing Parallelism: Predicated Instructions

Techniques such as loop unrolling, software pipelining, and trace scheduling can be used to increase the amount of parallelism available when the behavior of branches is fairly predictable at compile time. When the behavior of branches is not well known, compiler techniques alone may not be able to uncover much ILP. In such cases, the control dependences may severely limit the amount of parallelism that can be exploited. To overcome these problems, an architect can extend the instruction set to include *conditional* or *predicated instructions*. Such instructions can be used to eliminate branches, converting a control dependence into a data dependence and potentially improving performance. Such approaches are useful with either the hardware-intensive schemes in Chapter 2 or the software-intensive approaches discussed in this appendix, since in both cases, predication can be used to eliminate branches.

The concept behind conditional instructions is quite simple: An instruction refers to a condition, which is evaluated as part of the instruction execution. If the condition is true, the instruction is executed normally; if the condition is false, the execution continues as if the instruction were a no-op. Many newer architectures include some form of conditional instructions. The most common example of such an instruction is conditional move, which moves a value from one register to another if the condition is true. Such an instruction can be used to completely eliminate a branch in simple code sequences.

Example Consider the following code:

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

Assuming that registers R1, R2, and R3 hold the values of A, S, and T, respectively, show the code for this statement with the branch and with the conditional move.

Answer The straightforward code using a branch for this statement is (remember that we are assuming normal rather than delayed branches)

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

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

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

The conditional instruction allows us to convert the control dependence present in the branch-based code sequence to a data dependence. (This transformation is also used for vector computers, where it is called *if conversion*.) For a pipelined processor, this moves the place where the dependence must be resolved from near the front of the pipeline, where it is resolved for branches, to the end of the pipeline, where the register write occurs.

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

In the example above or in the compilation of absolute value, conditional moves are used to change a control dependence into a data dependence. This enables us to eliminate the branch and possibly improve the pipeline behavior. As issue rates increase, designers are faced with one of two choices: execute multiple branches per clock cycle or find a method to eliminate branches to avoid this requirement. Handling multiple branches per clock is complex, since one branch must be control dependent on the other. The difficulty of accurately predicting two branch outcomes, updating the prediction tables, and executing the correct sequence has so far caused most designers to avoid processors that execute multiple branches per clock. Conditional moves and predicated instructions provide a way of reducing the branch pressure. In addition, a conditional move can often eliminate a branch that is hard to predict, increasing the potential gain.

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

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

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

Example Here is a code sequence for a two-issue superscalar that can issue a combination of one memory reference and one ALU operation, or a branch by itself, every cycle:

First instruction slot	Second instruction slot
LW R1,40(R2)	ADD R3,R4,R5
	ADD R6,R3,R7
BEQZ R10,L	
LW R8,0(R10)	
LW R9,0(R8)	

This sequence wastes a memory operation slot in the second cycle and will incur a data dependence stall if the branch is not taken, since the second LW after the branch depends on the prior load. Show how the code can be improved using a predicated form of LW.

Answer Call the predicated version load word LWC and assume the load occurs unless the third operand is 0. The LW immediately following the branch can be converted to an LWC and moved up to the second issue slot:

First instruction slot	Second instruction slot
LW R1,40(R2)	ADD R3,R4,R5
LWC R8,0(R10),R10	ADD R6,R3,R7
BEQZ R10,L	
LW R9,0(R8)	

This improves the execution time by several cycles since it eliminates one instruction issue slot and reduces the pipeline stall for the last instruction in the sequence. Of course, if the compiler mispredicted the branch, the predicated instruction will have no effect and will not improve the running time. This is why the transformation is speculative.

If the sequence following the branch were short, the entire block of code might be converted to predicated execution and the branch eliminated.

When we convert an entire code segment to predicated execution or speculatively move an instruction and make it predicted, we remove a control dependence. Correct code generation and the conditional execution of predicated instructions ensure that we maintain the data flow enforced by the branch. To ensure that the exception behavior is also maintained, a predicated instruction must not generate an exception if the predicate is false. The property of not caus-

ing exceptions is quite critical, as the previous example shows: If register R10 contains zero, the instruction LW R8,0(R10) executed unconditionally is likely to cause a protection exception, and this exception should not occur. Of course, if the condition is satisfied (i.e., R10 is not zero), the LW may still cause a legal and resumable exception (e.g., a page fault), and the hardware must take the exception when it knows that the controlling condition is true.

The major complication in implementing predicated instructions is deciding when to annul an instruction. Predicated instructions may either be annulled during instruction issue or later in the pipeline before they commit any results or raise an exception. Each choice has a disadvantage. If predicated instructions are annulled early in the pipeline, the value of the controlling condition must be known early to prevent a stall for a data hazard. Since data-dependent branch conditions, which tend to be less predictable, are candidates for conversion to predicated execution, this choice can lead to more pipeline stalls. Because of this potential for data hazard stalls, no design with predicated execution (or conditional move) annuls instructions early. Instead, all existing processors annul instructions later in the pipeline, which means that annulled instructions will consume functional unit resources and potentially have a negative impact on performance. A variety of other pipeline implementation techniques, such as forwarding, interact with predicated instructions, further complicating the implementation.

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

- Predicated instructions that are annulled (i.e., whose conditions are false) still take some processor resources. An annulled predicated instruction requires fetch resources at a minimum, and in most processors functional unit execution time. Therefore, moving an instruction across a branch and making it conditional will slow the program down whenever the moved instruction would not have been normally executed. Likewise, predicating a control-dependent portion of code and eliminating a branch may slow down the processor if that code would not have been executed. An important exception to these situations occurs when the cycles used by the moved instruction when it is not performed would have been idle anyway (as in the earlier superscalar example). Moving an instruction across a branch or converting a code segment to predicated execution is essentially speculating on the outcome of the branch. Conditional instructions make this easier but do not eliminate the execution time taken by an incorrect guess. In simple cases, where we trade a conditional move for a branch and a move, using conditional moves or predication is almost always better. When longer code sequences are made conditional, the benefits are more limited.
- Predicated instructions are most useful when the predicate can be evaluated early. If the condition evaluation and predicated instructions cannot be separated (because of data dependences in determining the condition), then a con-

ditional instruction may result in a stall for a data hazard. With branch prediction and speculation, such stalls can be avoided, at least when the branches are predicted accurately.

- The use of conditional instructions can be limited when the control flow involves more than a simple alternative sequence. For example, moving an instruction across multiple branches requires making it conditional on both branches, which requires two conditions to be specified or requires additional instructions to compute the controlling predicate. If such capabilities are not present, the overhead of if conversion will be larger, reducing its advantage.
- Conditional instructions may have some speed penalty compared with unconditional instructions. This may show up as a higher cycle count for such instructions or a slower clock rate overall. If conditional instructions are more expensive, they will need to be used judiciously.

For these reasons, many architectures have included a few simple conditional instructions (with conditional move being the most frequent), but only a few architectures include conditional versions for the majority of the instructions. The MIPS, Alpha, PowerPC, SPARC, and Intel x86 (as defined in the Pentium processor) all support conditional move. The IA-64 architecture supports full predication for all instructions, as we will see in Section G.6.

G.5 Hardware Support for Compiler Speculation

As we saw in Chapter 2, many programs have branches that can be accurately predicted at compile time either from the program structure or by using a profile. In such cases, the compiler may want to speculate either to improve the scheduling or to increase the issue rate. Predicated instructions provide one method to speculate, but they are really more useful when control dependences can be completely eliminated by if conversion. In many cases, we would like to move speculated instructions not only before the branch, but before the condition evaluation, and predication cannot achieve this.

To speculate ambitiously requires three capabilities:

1. The ability of the compiler to find instructions that, with the possible use of register renaming, can be speculatively moved and not affect the program data flow
2. The ability to ignore exceptions in speculated instructions, until we know that such exceptions should really occur
3. The ability to speculatively interchange loads and stores, or stores and stores, which may have address conflicts

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

Hardware Support for Preserving Exception Behavior

To speculate ambitiously, we must be able to move any type of instruction and still preserve its exception behavior. The key to being able to do this is to observe that the results of a speculated sequence that is mispredicted will not be used in the final computation, and such a speculated instruction should not cause an exception.

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

1. The hardware and operating system cooperatively ignore exceptions for speculative instructions. As we will see later, this approach preserves exception behavior for correct programs, but not for incorrect ones. This approach may be viewed as unacceptable for some programs, but it has been used, under program control, as a "fast mode" in several processors.
2. Speculative instructions that never raise exceptions are used, and checks are introduced to determine when an exception should occur.
3. A set of status bits, called *poison bits,* are attached to the result registers written by speculated instructions when the instructions cause exceptions. The poison bits cause a fault when a normal instruction attempts to use the register.
4. A mechanism is provided to indicate that an instruction is speculative, and the hardware buffers the instruction result until it is certain that the instruction is no longer speculative.

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

Exceptions that indicate a program error should not occur in correct programs, and the result of a program that gets such an exception is not well defined, except perhaps when the program is running in a debugging mode. If such exceptions arise in speculated instructions, we cannot take the exception until we know that the instruction is no longer speculative.

In the simplest method for preserving exceptions, the hardware and the operating system simply handle all resumable exceptions when the exception occurs and simply return an undefined value for any exception that would cause termination. If the instruction generating the terminating exception was not speculative,

then the program is in error. Note that instead of terminating the program, the program is allowed to continue, although it will almost certainly generate incorrect results. If the instruction generating the terminating exception is speculative, then the program may be correct and the speculative result will simply be unused; thus, returning an undefined value for the instruction cannot be harmful. This scheme can never cause a correct program to fail, no matter how much speculation is done. An incorrect program, which formerly might have received a terminating exception, will get an incorrect result. This is acceptable for some programs, assuming the compiler can also generate a normal version of the program, which does not speculate and can receive a terminating exception.

Example Consider the following code fragment from an if-then-else statement of the form

```
if (A==0) A = B; else A = A+4;
```

where A is at 0(R3) and B is at 0(R2):

```
        LD      R1,0(R3)    ;load A
        BNEZ    R1,L1       ;test A
        LD      R1,0(R2)    ;then clause
        J       L2          ;skip else
L1:     DADDI   R1,R1,#4    ;else clause
L2:     SD      R1,0(R3)    ;store A
```

Assume the then clause is *almost always* executed. Compile the code using compiler-based speculation. Assume R14 is unused and available.

Answer Here is the new code:

```
        LD      R1,0(R3)     ;load A
        LD      R14,0(R2)    ;speculative load B
        BEQZ    R1,L3        ;other branch of the if
        DADDI   R14,R1,#4    ;the else clause
L3:     SD      R14,0(R3)    ;nonspeculative store
```

The then clause is completely speculated. We introduce a temporary register to avoid destroying R1 when B is loaded; if the load is speculative, R14 will be useless. After the entire code segment is executed, A will be in R14. The else clause could have also been compiled speculatively with a conditional move, but if the branch is highly predictable and low cost, this might slow the code down, since two extra instructions would always be executed as opposed to one branch.

In such a scheme, it is not necessary to know that an instruction is speculative. Indeed, it is helpful only when a program is in error and receives a terminating exception on a normal instruction; in such cases, if the instruction were not marked as speculative, the program could be terminated.

In this method for handling speculation, as in the next one, renaming will often be needed to prevent speculative instructions from destroying live values. Renaming is usually restricted to register values. Because of this restriction, the targets of stores cannot be destroyed and stores cannot be speculative. The small number of registers and the cost of spilling will act as one constraint on the amount of speculation. Of course, the major constraint remains the cost of executing speculative instructions when the compiler's branch prediction is incorrect.

A second approach to preserving exception behavior when speculating introduces speculative versions of instructions that do not generate terminating exceptions and instructions to check for such exceptions. This combination preserves the exception behavior exactly.

Example Show how the previous example can be coded using a speculative load (sLD) and a speculation check instruction (SPECCK) to completely preserve exception behavior. Assume R14 is unused and available.

Answer Here is the code that achieves this:

```
        LD      R1,0(R3)    ;load A
        sLD     R14,0(R2)   ;speculative, no termination
        BNEZ    R1,L1       ;test A
        SPECCK  0(R2)       ;perform speculation check
        J       L2          ;skip else
L1:     DADDI   R14,R1,#4   ;else clause
L2:     SD      R14,0(R3)   ;store A
```

Notice that the speculation check requires that we maintain a basic block for the then case. If we had speculated only a portion of the then case, then a basic block representing the then case would exist in any event. More importantly, notice that checking for a possible exception requires extra code.

A third approach for preserving exception behavior tracks exceptions as they occur but postpones any terminating exception until a value is actually used, preserving the occurrence of the exception, although not in a completely precise fashion. The scheme is simple: A poison bit is added to every register, and another bit is added to every instruction to indicate whether the instruction is speculative. The poison bit of the destination register is set whenever a speculative instruction results in a terminating exception; all other exceptions are handled immediately. If a speculative instruction uses a register with a poison bit turned on, the destination register of the instruction simply has its poison bit turned on. If a normal instruction attempts to use a register source with its poison bit turned on, the instruction causes a fault. In this way, any program that would have generated an exception still generates one, albeit at the first instance where a result is used by an instruction that is not speculative. Since poison bits exist only

on register values and not memory values, stores are never speculative and thus trap if either operand is "poison."

Example Consider the code fragment from page G-29 and show how it would be compiled with speculative instructions and poison bits. Show where an exception for the speculative memory reference would be recognized. Assume R14 is unused and available.

Answer Here is the code (an s preceding the opcode indicates a speculative instruction):

```
        LD      R1,0(R3)     ;load A
        sLD     R14,0(R2)    ;speculative load B
        BEQZ    R1,L3        ;
        DADDI   R14,R1,#4    ;
L3:     SD      R14,0(R3)    ;exception for speculative LW
```

If the speculative sLD generates a terminating exception, the poison bit of R14 will be turned on. When the nonspeculative SW instruction occurs, it will raise an exception if the poison bit for R14 is on.

One complication that must be overcome is how the OS saves the user registers on a context switch if the poison bit is set. A special instruction is needed to save and reset the state of the poison bits to avoid this problem.

The fourth and final approach listed earlier relies on a hardware mechanism that operates like a reorder buffer. In such an approach, instructions are marked by the compiler as speculative and include an indicator of how many branches the instruction was speculatively moved across and what branch action (taken/not taken) the compiler assumed. This last piece of information basically tells the hardware the location of the code block where the speculated instruction originally was. In practice, most of the benefit of speculation is gained by allowing movement across a single branch, and, thus, only 1 bit saying whether the speculated instruction came from the taken or not taken path is required. Alternatively, the original location of the speculative instruction is marked by a *sentinel,* which tells the hardware that the earlier speculative instruction is no longer speculative and values may be committed.

All instructions are placed in a reorder buffer when issued and are forced to commit in order, as in a hardware speculation approach. (Notice, though, that no actual speculative branch prediction or dynamic scheduling occurs.) The reorder buffer tracks when instructions are ready to commit and delays the "write-back" portion of any speculative instruction. Speculative instructions are not allowed to commit until the branches that have been speculatively moved over are also ready to commit, or, alternatively, until the corresponding sentinel is reached. At that point, we know whether the speculated instruction should have been executed or not. If it should have been executed and it generated a terminating exception, then we know that the program should be terminated. If the instruction should not

have been executed, then the exception can be ignored. Notice that the compiler, rather than the hardware, has the job of register renaming to ensure correct usage of the speculated result, as well as correct program execution.

Hardware Support for Memory Reference Speculation

Moving loads across stores is usually done when the compiler is certain the addresses do not conflict. As we saw with the examples in Section 2.2, such transformations are critical to reducing the critical path length of a code segment. To allow the compiler to undertake such code motion when it cannot be absolutely certain that such a movement is correct, a special instruction to check for address conflicts can be included in the architecture. The special instruction is left at the original location of the load instruction (and acts like a guardian), and the load is moved up across one or more stores.

When a speculated load is executed, the hardware saves the address of the accessed memory location. If a subsequent store changes the location before the check instruction, then the speculation has failed. If the location has not been touched, then the speculation is successful. Speculation failure can be handled in two ways. If only the load instruction was speculated, then it suffices to redo the load at the point of the check instruction (which could supply the target register in addition to the memory address). If additional instructions that depended on the load were also speculated, then a fix-up sequence that reexecutes all the speculated instructions starting with the load is needed. In this case, the check instruction specifies the address where the fix-up code is located.

In this section we have seen a variety of hardware assist mechanisms. Such mechanisms are key to achieving good support with the compiler-intensive approaches of Chapter 2 and this appendix. In addition, several of them can be easily integrated in the hardware-intensive approaches of Chapter 2 and provide additional benefits.

G.6 The Intel IA-64 Architecture and Itanium Processor

This section is an overview of the Intel IA-64 architecture, the most advanced VLIW-style processor, and its implementation in the Itanium processor.

The Intel IA-64 Instruction Set Architecture

The IA-64 is a RISC-style, register-register instruction set, but with many novel features designed to support compiler-based exploitation of ILP. Our focus here is on the unique aspects of the IA-64 ISA. Most of these aspects have been discussed already in this appendix, including predication, compiler-based parallelism detection, and support for memory reference speculation.

When they announced the IA-64 architecture, HP and Intel introduced the term EPIC (Explicitly Parallel Instruction Computer) to distinguish this new architectural approach from the earlier VLIW architectures and from other RISC architectures. Although VLIW and EPIC architectures share many features, the EPIC approach includes several concepts that extend the earlier VLIW approach. These extensions fall into two main areas:

1. EPIC has greater flexibility in indicating parallelism among instructions and in instruction formats. Rather than relying on a fixed instruction format where all operations in the instruction must be capable of being executed in parallel and where the format is completely rigid, EPIC uses explicit indicators of possible instruction dependence as well as a variety of instruction formats. This EPIC approach can express parallelism more flexibly than the more rigid VLIW method and can reduce the increases in code size caused by the typically inflexible VLIW instruction format.
2. EPIC has more extensive support for software speculation than the earlier VLIW schemes that had only minimal support.

In addition, the IA-64 architecture includes a variety of features to improve performance, such as register windows and a rotating floating-point register stack.

The IA-64 Register Model

The components of the IA-64 register state are

- 128 64-bit general-purpose registers, which as we will see shortly are actually 65 bits wide
- 128 82-bit floating-point registers, which provide two extra exponent bits over the standard 80-bit IEEE format
- 64 1-bit predicate registers
- 8 64-bit branch registers, which are used for indirect branches
- a variety of registers used for system control, memory mapping, performance counters, and communication with the OS

The integer registers are configured to help accelerate procedure calls using a register stack mechanism similar to that developed in the Berkeley RISC-I processor and used in the SPARC architecture. Registers 0–31 are always accessible and are addressed as 0–31. Registers 32–128 are used as a register stack, and each procedure is allocated a set of registers (from 0 to 96) for its use. The new register stack frame is created for a called procedure by renaming the registers in hardware; a special register called the current frame pointer (CFM) points to the set of registers to be used by a given procedure. The frame consists of two parts: the local area and the output area. The local area is used for local storage, while the output area is used to pass values to any called procedure. The `alloc` instruction specifies the size of these areas. Only the integer registers have register stack support.

On a procedure call, the CFM pointer is updated so that R32 of the called procedure points to the first register of the output area of the called procedure. This update enables the parameters of the caller to be passed into the addressable registers of the callee. The callee executes an `alloc` instruction to allocate both the number of required local registers, which include the output registers of the caller, and the number of output registers needed for parameter passing to a called procedure. Special load and store instructions are available for saving and restoring the register stack, and special hardware (called the *register stack engine*) handles overflow of the register stack.

In addition to the integer registers, there are three other sets of registers: the floating-point registers, the predicate registers, and the branch registers. The floating-point registers are used for floating-point data, and the branch registers are used to hold branch destination addresses for indirect branches. The predication registers hold predicates, which control the execution of predicated instructions; we describe the predication mechanism later in this section.

Both the integer and floating-point registers support register rotation for registers 32–128. Register rotation is designed to ease the task of allocating registers in software-pipelined loops, a problem that we discussed in Section G.3. In addition, when combined with the use of predication, it is possible to avoid the need for unrolling and for separate prologue and epilogue code for a software-pipelined loop. This capability reduces the code expansion incurred to use software pipelining and makes the technique usable for loops with smaller numbers of iterations, where the overheads would traditionally negate many of the advantages.

Instruction Format and Support for Explicit Parallelism

The IA-64 architecture is designed to achieve the major benefits of a VLIW approach—implicit parallelism among operations in an instruction and fixed formatting of the operation fields—while maintaining greater flexibility than a VLIW normally allows. This combination is achieved by relying on the compiler to detect ILP and schedule instructions into parallel instruction slots, but adding flexibility in the formatting of instructions and allowing the compiler to indicate when an instruction cannot be executed in parallel with its successors.

The IA-64 architecture uses two different concepts to achieve the benefits of implicit parallelism and ease of instruction decode. Implicit parallelism is achieved by placing instructions into *instruction groups,* while the fixed formatting of multiple instructions is achieved through the introduction of a concept called a *bundle,* which contains three instructions. Let's start by defining an instruction group.

An instruction group is a sequence of consecutive instructions with no register data dependences among them (there are a few minor exceptions). All the instructions in a group could be executed in parallel, if sufficient hardware resources existed and if any dependences through memory were preserved. An instruction group can be arbitrarily long, but the compiler must *explicitly* indicate

Execution unit slot	Instruction type	Instruction description	Example instructions
I-unit	A	Integer ALU	add, subtract, and, or, compare
	I	Non-ALU integer	integer and multimedia shifts, bit tests, moves
M-unit	A	Integer ALU	add, subtract, and, or, compare
	M	Memory access	Loads and stores for integer/FP registers
F-unit	F	Floating point	Floating-point instructions
B-unit	B	Branches	Conditional branches, calls, loop branches
L + X	L + X	Extended	Extended immediates, stops and no-ops

Figure G.6 The five execution unit slots in the IA-64 architecture and what instructions types they may hold are shown. A-type instructions, which correspond to integer ALU instructions, may be placed in either an I-unit or M-unit slot. L + X slots are special, as they occupy two instruction slots; L + X instructions are used to encode 64-bit immediates and a few special instructions. L + X instructions are executed either by the I-unit or the B-unit.

the boundary between one instruction group and another. This boundary is indicated by placing a *stop* between two instructions that belong to different groups. To understand how stops are indicated, we must first explain how instructions are placed into bundles.

IA-64 instructions are encoded in bundles, which are 128 bits wide. Each bundle consists of a 5-bit template field and three instructions, each 41 bits in length. (Actually, the 41-bit quantities are not truly instructions, since they can only be interpreted in conjunction with the template field. The name *syllable* is sometimes used for these operations. For simplicity, we will continue to use the term "instruction.") To simplify the decoding and instruction issue process, the template field of a bundle specifies what types of execution units each instruction in the bundle requires. Figure G.6 shows the five different execution unit types and describes what instruction classes they may hold, together with some examples.

The 5-bit template field within each bundle describes *both* the presence of any stops associated with the bundle and the execution unit type required by each instruction within the bundle. Figure G.7 shows the possible formats that the template field encodes and the position of any stops it specifies. The bundle formats can specify only a subset of all possible combinations of instruction types and stops. To see how the bundle works, let's consider an example.

Example Unroll the array increment example, `x[i] = x[i] + s` (introduced on page 305), seven times (see page 317 for the unrolled code) and place the instructions into bundles, first ignoring pipeline latencies (to minimize the number of bundles) and then scheduling the code to minimize stalls. In scheduling the code assume one

Template	Slot 0	Slot 1	Slot 2
0	M	I	I
1	M	I	I
2	M	I	I
3	M	I	I
4	M	L	X
5	M	L	X
8	M	M	I
9	M	M	I
10	M	M	I
11	M	M	I
12	M	F	I
13	M	F	I
14	M	M	F
15	M	M	F
16	M	I	B
17	M	I	B
18	M	B	B
19	M	B	B
22	B	B	B
23	B	B	B
24	M	M	B
25	M	M	B
28	M	F	B
29	M	F	B

Figure G.7 The 24 possible template values (8 possible values are reserved) and the instruction slots and stops for each format. Stops are indicated by heavy lines and may appear within and/or at the end of the bundle. For example, template 9 specifies that the instruction slots are M, M, and I (in that order) and that the only stop is between this bundle and the next. Template 11 has the same type of instruction slots but also includes a stop after the first slot. The L + X format is used when slot 1 is L and slot 2 is X.

bundle executes per clock and that any stalls cause the entire bundle to be stalled. Use the pipeline latencies from Figure 2.2. Use MIPS instruction mnemonics for simplicity.

Answer The two different versions are shown in Figure G.8. Although the latencies are different from those in Itanium, the most common bundle, MMF, must be issued by itself in Itanium, just as our example assumes.

Bundle template	Slot 0	Slot 1	Slot 2	Execute cycle (1 bundle/cycle)
9: M M I	L.D F0,0(R1)	L.D F6,-8(R1)		1
14: M M F	L.D F10,-16(R1)	L.D F14,-24(R1)	ADD.D F4,F0,F2	3
15: M M F	L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F8,F6,F2	4
15: M M F	L.D F26,-48(R1)	S.D F4,0(R1)	ADD.D F12,F10,F2	6
15: M M F	S.D F8,-8(R1)	S.D F12,-16(R1)	ADD.D F16,F14,F2	9
15: M M F	S.D F16,-24(R1)		ADD.D F20,F18,F2	12
15: M M F	S.D F20,-32(R1)		ADD.D F24,F22,F2	15
15: M M F	S.D F24,-40(R1)		ADD.D F28,F26,F2	18
16: M I B	S.D F28,-48(R1)	DADDUI R1,R1,#-56	BNE R1,R2,Loop	21

(a) The code scheduled to minimize the number of bundles

Bundle template	Slot 0	Slot 1	Slot 2	Execute cycle (1 bundle/cycle)
8: M M I	L.D F0,0(R1)	L.D F6,-8(R1)		1
9: M M I	L.D F10,-16(R1)	L.D F14,-24(R1)		2
14: M M F	L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	3
14: M M F	L.D F26,-48(R1)		ADD.D F8,F6,F2	4
15: M M F			ADD.D F12,F10,F2	5
14: M M F		S.D F4,0(R1)	ADD.D F16,F14,F2	6
14: M M F		S.D F8,-8(R1)	ADD.D F20,F18,F2	7
15: M M F		S.D F12,-16(R1)	ADD.D F24,F22,F2	8
14: M M F		S.D F16,-24(R1)	ADD.D F28,F26,F2	9
9: M M I	S.D F20,-32(R1)	S.D F24,-40(R1)		11
16: M I B	S.D F28,-48(R1)	DADDUI R1,R1,#-56	BNE R1,R2,Loop	12

(b) The code scheduled to minimize the number of cycles assuming one bundle executed per cycle

Figure G.8 The IA-64 instructions, including bundle bits and stops, for the unrolled version of `x[i] = x[i] + s`, when unrolled seven times and scheduled (a) to minimize the number of instruction bundles and (b) to minimize the number of cycles (assuming that a hazard stalls an entire bundle). Blank entries indicate unused slots, which are encoded as no-ops. The absence of stops indicates that some bundles could be executed in parallel. Minimizing the number of bundles yields 9 bundles versus the 11 needed to minimize the number of cycles. The scheduled version executes in just over half the number of cycles. Version (a) fills 85% of the instruction slots, while (b) fills 70%. The number of empty slots in the scheduled code and the use of bundles may lead to code sizes that are much larger than other RISC architectures. Note that the branch in the last bundle in both sequences depends on the DADD in the same bundle. In the IA-64 instruction set, this sequence would be coded as a setting of a predication register and a branch that would be predicated on that register. Normally, such dependent operations could not occur in the same bundle, but this case is one of the exceptions mentioned earlier.

Instruction Set Basics

Before turning to the special support for speculation, we briefly discuss the major instruction encodings and survey the instructions in each of the five primary instruction classes (A, I, M, F, and B). Each IA-64 instruction is 41 bits in length. The high-order 4 bits, together with the bundle bits that specify the execution unit slot, are used as the major opcode. (That is, the 4-bit opcode field is reused across the execution field slots, and it is appropriate to think of the opcode as being 4 bits plus the M, F, I, B, L + X designation.) The low-order 6 bits of every instruction are used for specifying the predicate register that guards the instruction (see the next subsection).

Figure G.9 summarizes most of the major instruction formats, other than the multimedia instructions, and gives examples of the instructions encoded for each format.

Predication and Speculation Support

The IA-64 architecture provides comprehensive support for predication: Nearly every instruction in the IA-64 architecture can be predicated. An instruction is predicated by specifying a predicate register, whose identity is placed in the lower 6 bits of each instruction field. Because nearly all instructions can be predicated, both if conversion and code motion have lower overhead than they would with only limited support for conditional instructions. One consequence of full predication is that a conditional branch is simply a branch with a guarding predicate!

Predicate registers are set using compare or test instructions. A compare instruction specifies one of ten different comparison tests and two predicate registers as destinations. The two predicate registers are written either with the result of the comparison (0 or 1) and the complement, or with some logical function that combines the two tests (such as `and`) and the complement. This capability allows multiple comparisons to be done in one instruction.

Speculation support in the IA-64 architecture consists of separate support for control speculation, which deals with deferring exception for speculated instructions, and memory reference speculation, which supports speculation of load instructions.

Deferred exception handling for speculative instructions is supported by providing the equivalent of poison bits. For the GPRs, these bits are called NaTs (Not a Thing), and this extra bit makes the GPRs effectively 65 bits wide. For the FP registers this capability is obtained using a special value, NaTVal (Not a Thing Value); this value is encoded using a significand of 0 and an exponent outside of the IEEE range. Only speculative load instructions generate such values, but all instructions that do not affect memory will cause a NaT or NaTVal to be propagated to the result register. (There are both speculative and nonspeculative loads; the latter can only raise immediate exceptions and cannot defer them.)

Instruction type	Number of formats	Representative instructions	Extra opcode bits	GPRs/ FPRs	Immediate bits	Other/comment
A	8	add, subtract, and, or	9	3	0	
		shift left and add	7	3	0	2-bit shift count
		ALU immediates	9	2	8	
		add immediate	3	2	14	
		add immediate	0	2	22	
		compare	4	2	0	2 predicate register destinations
		compare immediate	3	1	8	2 predicate register destinations
I	29	shift R/L variable	9	3	0	Many multimedia instructions use this format.
		test bit	6	3	6-bit field specifier	2 predicate register destinations
		move to BR	6	1	9-bit branch predict	branch register specifier
M	46	integer/FP load and store, line prefetch	10	2	0	speculative/ nonspeculative
		integer/FP load and store, and line prefetch and post-increment by immediate	9	2	8	speculative/ nonspeculative
		integer/FP load prefetch and register postincrement	10	3		speculative/ nonspeculative
		integer/FP speculation check	3	1	21 in two fields	
B	9	PC-relative branch, counted branch	7	0	21	
		PC-relative call	4	0	21	1 branch register
F	15	FP arithmetic	2	4		
		FP compare	2	2		2 6-bit predicate regs
L + X	4	move immediate long	2	1	64	

Figure G.9 A summary of some of the instruction formats of the IA-64 ISA. The major opcode bits and the guarding predication register specifier add 10 bits to every instruction. The number of formats indicated for each instruction class in the second column (a total of 111) is a strict interpretation: A different use of a field, even of the same size, is considered a different format. The number of formats that actually have *different field sizes* is one-third to one-half as large. Some instructions have unused bits that are reserved; we have not included those in this table. Immediate bits include the sign bit. The branch instructions include prediction bits, which are used when the predictor does not have a valid prediction. Only one of the many formats for the multimedia instructions is shown in this table.

Floating-point exceptions are not handled through this mechanism, but use floating-point status registers to record exceptions.

A deferred exception can be resolved in two different ways. First, if a nonspeculative instruction, such as a store, receives a NaT or NaTVal as a source operand, it generates an immediate and unrecoverable exception. Alternatively, a `chk.s` instruction can be used to detect the presence of NaT or NaTVal and branch to a routine designed by the compiler to recover from the speculative operation. Such a recovery approach makes more sense for memory reference speculation.

The inability to store the contents of instructions with NaT or NaTVal set would make it impossible for the OS to save the state of the processor. Thus, IA-64 includes special instructions to save and restore registers that do not cause an exception for a NaT or NaTVal and also save and restore the NaT bits.

Memory reference support in the IA-64 uses a concept called *advanced loads*. An advanced load is a load that has been speculatively moved above store instructions on which it is potentially dependent. To speculatively perform a load, the `ld.a` (for advanced load) instruction is used. Executing this instruction creates an entry in a special table, called the *ALAT*. The ALAT stores both the register destination of the load and the address of the accessed memory location. When a store is executed, an associative lookup against the active ALAT entries is performed. If there is an ALAT entry with the same memory address as the store, the ALAT entry is marked as invalid.

Before any nonspeculative instruction (i.e., a store) uses the value generated by an advanced load or a value derived from the result of an advanced load, an explicit check is required. The check specifies the destination register of the advanced load. If the ALAT for that register is still valid, the speculation was legal and the only effect of the check is to clear the ALAT entry. If the check fails, the action taken depends on which of two different types of checks was employed. The first type of check is an instruction `ld.c`, which simply causes the data to be reloaded from memory at that point. An `ld.c` instruction is used when *only* the load is advanced. The alternative form of a check, `chk.a`, specifies the address of a fix-up routine that is used to reexecute the load *and any other* speculated code that depended on the value of the load.

The Itanium 2 Processor

The Itanium 2 processor is the second implementation of the IA-64 architecture. The first version, Itanium 1, became available in 2001 with an 800 MHz clock. The Itanium 2, first delivered in 2003, has a maximum clock rate in 2005 of 1.6 GHz. The two processors are very similar, with some differences in the pipeline structure and greater differences in the memory hierarchies. The Itanium 2 is about four times faster than the Itanium 1. This performance improvement comes from a doubling of the clock rate, a more aggressive memory hierarchy, additional functional units that improve instruction throughput, more complete

bypassing, a shorter pipeline that reduces some stalls, and a more mature compiler system. During roughly the same period that elapsed from the Itanium 1 to Itanium 2, the Pentium processors improved by slightly more than a factor of three. The greater improvement for the Itanium is reasonable given the novelty of the architecture and software system versus the more established IA-32 implementations.

The Itanium 2 can fetch and issue two bundles, or up to six instructions, per clock. The Itanium 2 uses a three-level memory hierarchy all on-chip. The first level uses split instruction and data caches, each 16 KB; floating-point data are not placed in the first-level cache. The second and third levels are unified caches of 256 KB and of 3 MB to 9 MB, respectively.

Functional Units and Instruction Issue

There are 11 functional units in the Itanium 2 processor: two I-units, four M-units (two for loads and two for stores), three B-units, and two F-units. All the functional units are pipelined. Figure G.10 gives the pipeline latencies for some typical instructions. In addition, when a result is bypassed from one unit to another, there is usually at least one additional cycle of delay.

Itanium 2 can issue up to six instructions per clock from two bundles. In the worst case, if a bundle is split when it is issued, the hardware could see as few as four instructions: one from the first bundle to be executed and three from the second bundle. Instructions are allocated to functional units based on the bundle bits, ignoring the presence of no-ops or predicated instructions with untrue predicates. In addition, when issue to a functional unit is blocked because the next instruction to be issued needs an already committed unit, the resulting bundle is split. A split bundle still occupies one of the two bundle slots, even if it has only one instruction remaining.

Instruction	Latency
Integer load	1
Floating-point load	5–9
Correctly predicted taken branch	0–3
Mispredicted branch	6
Integer ALU operations	0
FP arithmetic	4

Figure G.10 The latency of some typical instructions on Itanium 2. The latency is defined as the smallest number of intervening instructions between two dependent instructions. Integer load latency assumes a hit in the first-level cache. FP loads always bypass the primary cache, so the latency is equal to the access time of the second-level cache. There are some minor restrictions for some of the functional units, but these primarily involve the execution of infrequent instructions.

The Itanium 2 processor uses an eight-stage pipeline divided into four major parts:

- *Front-end (stages IPG and Rotate)*—Prefetches up to 32 bytes per clock (two bundles) into a prefetch buffer, which can hold up to eight bundles (24 instructions). Branch prediction is done using a multilevel adaptive predictor like those described in Chapter 2.
- *Instruction delivery (stages EXP and REN)*—Distributes up to six instructions to the 11 functional units. Implements register renaming for both rotation and register stacking.
- *Operand delivery (REG)*—Accesses the register file, performs register bypassing, accesses and updates a register scoreboard, and checks predicate dependences. The scoreboard is used to detect when individual instructions can proceed, so that a stall of one instruction (for example, due to an unpredictable event like a cache miss) in a bundle need not cause the entire bundle to stall. (As we saw in Figure G.8, stalling the entire bundle leads to poor performance unless the instructions are carefully scheduled.)
- *Execution (EXE, DET, and WRB)*—Executes instructions through ALUs and load-store units, detects exceptions and posts NaTs, retires instructions, and performs write back.

Both the Itanium 1 and the Itanium 2 have many of the features more commonly associated with the dynamically scheduled pipelines described in Chapter 2: dynamic branch prediction, register renaming, scoreboarding, a pipeline with a number of stages before execution (to handle instruction alignment, renaming, etc.), and several stages following execution to handle exception detection. Although these mechanisms are generally simpler than those in an advanced dynamically scheduled superscalar, the overall effect is that the Itanium processors, which rely much more on compiler technology, seem to be as complex as the dynamically scheduled processors we saw in Chapter 2!

One might ask why such features are included in a processor that relies primarily on compile time techniques for finding and exploiting parallelism. There are two main motivations. First, dynamic techniques are sometimes significantly better, and omitting them would hurt performance significantly. The inclusion of dynamic branch prediction is such a case.

Second, caches are absolutely necessary to achieve high performance, and with caches come cache misses, which are both unpredictable and which in current processors take a relatively long time. In the early VLIW processors, the entire processor would freeze when a cache miss occurred, retaining the lockstep parallelism initially specified by the compiler. Such an approach is totally unrealistic in a modern processor where cache misses can cost tens to hundreds of cycles. Allowing some instructions to continue while others are stalled, however, requires the introduction of some form of dynamic scheduling, in this case

scoreboarding. In addition, if a stall is likely to be long, then antidependences are likely to prevent much progress while waiting for the cache miss; hence, the Itanium implementations also introduce register renaming.

Itanium 2 Performance

Figure G.11 shows the performance of a 1.5 GHz Itanium 2 versus a Pentium 4, an AMD Athlon processor, and an IBM Power5 for five SPECint and five SPECfp benchmarks. Overall, the Itanium 2 is slightly slower than the Power5 for the full set of SPEC floating-point benchmarks and about 35% faster than the AMD Athlon or Pentium 4. On SPECint, the Itanium 2 is 15% faster than the Power5, while both the AMD Athlon and Pentium 4 are about 15% faster than the Itanium 2. As we explore in Chapter 3, the Itanium 2 and Power5 are much higher power and have larger die sizes. In fact, the Power5 contains two processors, only one of which is active during normal SPEC benchmarks, and still it has less than half the transistor count of the Itanium. If we were to reduce the die size, transistor count, and power of the Power5 by eliminating one of the processors, the Itanium would be by far the largest and highest-power processor. Section 3.6 contains a much more detailed comparison of these multiple-issue processors,

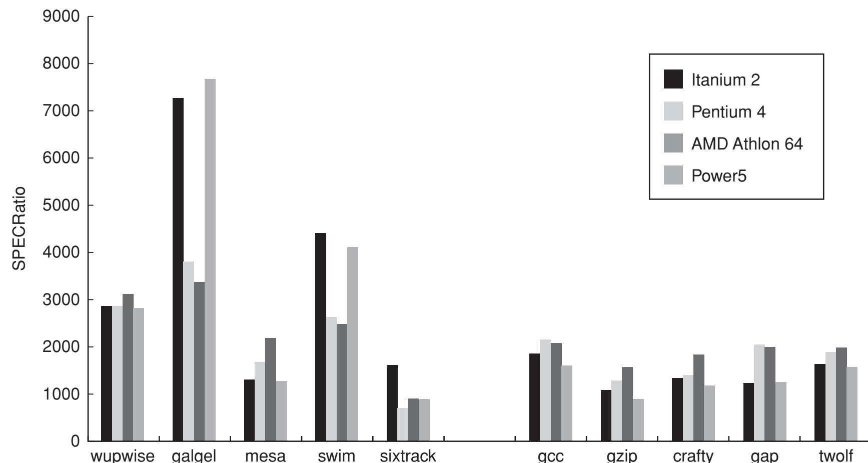

Figure G.11 The performance of four multiple-issue processors for five SPECfp and SPECint benchmarks. The clock rates of the five processors are Itanium 2 at 1.5 GHz, Pentium 4 Extreme Edition at 3.8 GHz, AMD Athlon 64 at 2.8 GHz, and the IBM Power5 at 1.9 GHz.

G.7 Concluding Remarks

When the design of the IA-64 architecture began, it was a joint effort of Hewlett-Packard and Intel and many of the designers had benefited from experience with early VLIW processors as well of years of research building on the early concepts. The clear goal for the IA-64 architecture was to achieve levels of ILP as good or better than what had been achieved with hardware-based approaches, while also allowing a much simpler hardware implementation. With a simpler hardware implementation, designers hoped that much higher clock rates could be achieved. Indeed when the IA-64 architecture and the first Itanium were announced, they were announced as the successor to the RISC approaches with clearly superior advantages.

Unfortunately, the practical reality has been quite different. The IA-64 and Itanium implementations appear to be at least as complicated as the dynamically based speculative processors, and neither approach has a significant and consistent performance advantage. The fact that the Itanium designs have also not been more power efficient has led to a situation where the Itanium design has been adopted by only a small number of customers primarily interested in FP performance.

Intel had planned for IA-64 to be its new 64-bit architecture as well. But the combination of its mediocre integer performance (especially in Itanium 1) and large die size, together with AMD's introduction of a 64-bit version of the IA-32 architecture, forced Intel to extend the address space of IA-32. The availability of a larger address space IA-32 processor with strong integer performance has further reduced the interest in IA-64 and Itanium. Most recently, Intel has introduced the name IPF to replace IA-64, since the former name made less sense once the older x86 architecture was extended to 64 bits.