tural Support for Programming Languages and Operating Systems (October), Boston, IEEE/ACM, 238–247.

Mahlke, S. A., R. E. Hank, J. E. McCormick, D. I. August, and W. W. Hwu [1995]. "A comparison of full and partial predicated execution support for ILP processors," *Proc. 22nd Annual Int'l Symposium on Computer Architecture* (June), Santa Margherita Ligure, Italy, 138–149.

McFarling, S., and J. Hennessy [1986]. "Reducing the cost of branches," *Proc. 13th Symposium on Computer Architecture* (June), Tokyo, 396–403.

Nicolau, A., and J. A. Fisher [1984]. "Measuring the parallelism available for very long instruction word architectures," *IEEE Trans. on Computers* C-33:11 (November), 968–976.

Rau, B. R. [1994]. "Iterative modulo scheduling: An algorithm for software pipelining loops," *Proc. 27th Annual Int'l Symposium on Microarchitecture* (November), San Jose, Calif., 63–74.

Rau, B. R., C. D. Glaeser, and R. L. Picard [1982]. "Efficient code generation for horizontal architectures: Compiler techniques and architectural support," *Proc. Ninth Symposium on Computer Architecture* (April), 131–139.

Rau, B. R., D. W. L. Yen, W. Yen, and R. A. Towle [1989]. "The Cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs," *IEEE Computers* 22:1 (January), 12–34.

Riseman, E. M., and C. C. Foster [1972]. "Percolation of code to enhance parallel dispatching and execution," *IEEE Trans. on Computers* C-21:12 (December), 1411–1415.

Smith, M. D., M. Horowitz, and M. S. Lam [1992]. "Efficient superscalar performance through boosting," *Proc. Fifth Conf. on Architectural Support for Programming Languages and Operating Systems* (October), Boston, IEEE/ACM, 248–259.

Thorlin, J. F. [1967]. "Code generation for PIE (parallel instruction execution) computers," *Proc. Spring Joint Computer Conf.*, 27.

Weiss, S., and J. E. Smith [1987]. "A study of scalar compilation techniques for pipelined supercomputers," *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems* (March), IEEE/ACM, Palo Alto, Calif., 105–109.

Wilson, R. P., and M. S. Lam [1995]. "Efficient context-sensitive pointer analysis for C programs," *Proc. ACM SIGPLAN'95 Conf. on Programming Language Design and Implementation*, La Jolla, Calif., June, 1–12.

Exercises

Solutions to the "starred" exercises appear in Appendix B.

✪ 4.1 [15/10] <4.1, A.4> If we assume the set of latencies in Figure 4.1 and that a result can always be forwarded, then a specific structure for some of the CPU pipeline is implied. Assume the CPU uses the standard five-stage IF/ID/EX/Mem/WB pipeline.

a. [15] <4.1, A.4> Using a style similar to that of Figures A.23 and A.31 in Appendix A, draw a block diagram showing only the implied portions of the pipeline. Label each component and data path in your diagram and show the number of clock cycles each functional unit requires.

b. [10] <4.1, A.4> For each functional unit in your part (a) diagram, which row(s) of Figure 4.1 provide the information to determine the number of clock cycles needed for that functional unit to complete its operation?

4.2 [15/15] <4.1> This chapter examines software approaches for exploiting instruction-level parallelism. This exercise asks how well software can find and exploit ILP. Chapter 3 presents hardware techniques for exposing and exploiting ILP. Exercise 3.2 is a reprise of this ILP analysis, but from a hardware perspective.

Consider the following four MIPS code fragments, each containing two instructions:

```
i.          DADDI R1,R1,#4
            LD R2,7(R1)

ii.         DADD R3,R1,R2
            SD 7(R1),R2

iii.        SD 7(R1),R2
            S.D 200(R7),F2

iv.         BEZ R1,place
            SD 7(R1),R1
```

a. [15] <4.1> For each code fragment (i)–(iv) identify each type of dependence that a compiler will find (a fragment may have no dependences) and describe what data flow, name reuse, or control structure causes the dependence.

b. [15] <4.1> Assuming nonspeculative execution, for each code fragment discuss whether a compiler could schedule the two instructions.

4.3 [20/15/15] <2.12, 4.1> Consider the simple loop in Section 4.1. Assume the number of iterations is unknown, but large.

a. [20] <4.1> Find the theoretically optimal number of unrollings using the latencies in Figure 4.1. *Hint:* Recall that you will need two loops: one unrolled and one not!

b. [15] <2.12, 4.1> What is the actual maximum number of times the simple loop in Section 4.1 can be unrolled using the given MIPS code? What is the limiting resource? Show how to increase the number of times the loop may be unrolled by transforming the MIPS code to make less intensive use of the limiting resource. How much does this transformation improve performance?

c. [15] <2.12, 4.1> For the MIPS instruction set, what additional parameters limit the number of times this loop can be unrolled? *Hint:* When you find one limiting parameter, assume that the resource it defines is unlimited, look for an additional parameter, and repeat as needed.

4.4 [15] <4.1> Section 4.1 presents a technique for unrolling loops where the unrolling factor is not statically known to be a factor of the number of loop iterations. For a factor of k, the technique constructs two consecutive loops that iterate (n mod k) and (n/k) times, respectively. Find a technique to use just a single loop containing the unrolled body iterated $\lceil n/k \rceil$ times. What restrictions are there on the use of this technique? When does this technique perform better than the general, two-consecutive-loops technique? Can a compiler employ this technique?

✪ 4.5 [15] <4.1> List all the dependences (output, anti, and true) in the following code fragment. Indicate whether the true dependences are loop carried or not. Show why the loop is not parallel.

```
for (i=2;i<100;i=i+1) {
      a[i] = b[i] + a[i];     /* S1 */
      c[i-1] = a[i] + d[i];   /* S2 */
      a[i-1] = 2 * b[i];      /* S3 */
      b[i+1] = 2 * b[i];      /* S4 */
}
```

4.6 [15] <4.1> Here is an unusual loop. First, list the dependences and then rewrite the loop so that it is parallel.

```
for (i=1;i<100;i=i+1) {
      a[i] = b[i] + c[i];      /* S1 */
      b[i] = a[i] + d[i];      /* S2 */
      a[i+1] = a[i] + e[i];    /* S3 */
}
```

✪ 4.7 [20/12] <4.1> The following loop is a dot product (assuming the running sum in F2 is initially 0) and contains a recurrence. Assume the pipeline latencies from Figure 4.1 and a 1-cycle delayed branch.

```
foo:   L.D       F0,0(R1)      ;load X[i]
       L.D       F4,0(R2)      ;load Y[i]
       MUL.D     F0,F0,F4      ;multiply X[i]*Y[i]
       ADD.D     F2,F0,F2      ;add sum = sum + X[i]*Y[i]
       DADDUI    R1,R1,#-8     ;decrement X index
       DADDUI    R2,R2,#-8     ;decrement Y index
       BNEZ      R1,foo        ;loop if not done
```

a. [20] <4.1> Assume a single-issue pipeline. Despite the fact that the loop is not parallel, it can be scheduled with no delays. Unroll the following loop a sufficient number of times to schedule it without any delays. Show the schedule after eliminating any redundant overhead instructions. *Hint:* An additional transformation of the code is needed to schedule without delay.

b. [12] <4.1> Show the schedule of the transformed code from part (a) for the processor in Figure 4.2. For an issue capability that is 100% greater, how much faster is the loop body?

4.8 [15/15] <4.1> The following loop computes Y[i] = a × X[i] + Y[i], the key step in a Gaussian elimination. Assume the pipeline latencies from Figure 4.1 and a 1-cycle delayed branch.

```
loop:  L.D       F0,0(R1)      ;load X[i]
       MUL.D     F0,F0,F2      ;multiply a*X[i]
       L.D       F4,0(R2)      ;load Y[i]
       ADD.D     F0,F0,F4      ;add a*X[i]+Y[i]
       S.D       0(R2),F0      ;store Y[i]
```

```
        DSUBUI     R1,R1,#8        ;decrement X index
        DSUBUI     R2,R2,#8        ;decrement Y index
        BNEZ       R1,loop         ;loop if not done
```

a. [15] <4.1> Assume a single-issue pipeline. Unroll the loop as many times as necessary to schedule it without any delays, collapsing the loop overhead instructions. Show the schedule. What is the execution time per element?

b. [15] <4.1> Assume a dual-issue processor as in Figure 4.2. Unroll the loop as many times as necessary to schedule it without any delays, collapsing the loop overhead instructions. Show the schedule. What is the execution time per element? How many instruction issue slots are unused?

4.9 [20/15/20/15/15] <1.5, 4.1, 4.3, 4.4> In this exercise, we look at how some software techniques can extract ILP in a common vector loop. The following loop is the so-called DAXPY loop (*d*ouble-precision *aX plus Y*; discussed in Appendix G) and the central operation in Gaussian elimination. The following code implements the DAXPY operation, Y = a × X + Y, for a vector length 100.

```
bar:    L.D        F2,0(R1)        ;load X(i)
        MUL.D      F4,F2,F0        ;multiply a*X(i)
        L.D        F6,0(R2)        ;load Y(i)
        ADD.D      F6,F4,F6        ;add a*X(i) + Y(i)
        S.D        0(R2),F6        ;store Y(i)
        DADDUI     R1,R1,#8        ;increment X index
        DADDUI     R2,R2,#8        ;increment Y index
        DSGTUI     R3,R1,#800      ;test if done
        BEQZ       R3,bar          ;loop if not done
```

For (a)–(e) assume the pipeline latencies from Figure 4.1 and a 1-cycle delayed branch that resolves in the ID stage. Assume that integer operations issue and complete in 1 clock cycle and that their results are fully bypassed.

a. [20] <1.5, 4.1> Assume a single-issue pipeline. Show how the loop would look both unscheduled by the compiler and after compiler scheduling for both floating-point operation and branch delays, including any stalls or idle clock cycles (see the example on page 305). What is the execution time per element of the result vector, Y, unscheduled and scheduled? How much faster must the clock be for processor hardware alone to match the performance improvement achieved by the scheduling compiler (neglect the possible increase in the number of clock cycles necessary for memory system access effects of higher processor clock speed on memory system performance)?

b. [15] <4.1> Assume a single-issue pipeline. Unroll the loop as many times as necessary to schedule it without any stalls, collapsing the loop overhead instructions. How many times must the loop be unrolled? Show the instruction schedule. What is the execution time per element of the result vector? What is the major contribution to the reduction in time per element?

c. [20] <4.3> Assume a VLIW processor with instructions that contain five operations, as shown in Figure 4.5. We will compare two degrees of loop unrolling. First, unroll the loop 4 times to extract ILP, and schedule it without any stalls (i.e., completely empty issue cycles), collapsing the loop overhead instructions, and then repeat the process but unroll the loop 10 times. Ignore the branch delay slot. Show the two schedules. What is the execution time per element of the result vector for each schedule? What percent of the operation slots are used in each schedule? How much does the size of the code differ between the two schedules? What is the total register demand for the two schedules?

d. [15] <4.4> Assume a family of VLIW processors designed for different price-performance points in the marketplace. The budget version of the processor has latencies greater than those in Figure 4.1. What will happen if the code for your answer to part (c) is executed on the low-cost processor? How could you eliminate any undesirable behavior?

e. [15] <4.4> Assume a single-issue pipeline. Show the schedule for a software-pipelined version of the DAXPY loop. You may omit the start-up and clean-up code. What is the execution time per element of the result vector?

4.10 [20/20] <4.4> In this exercise we finish the compiler code transformation started in the software-pipelining loop example on page 330.

a. [20] <4.4> Starting with the solution loop body given in the example, write code for the complete software-pipelined loop by adding the start-up and finish-up code. Assume that there will be a large number of iterations executed. You need not show code to initialize the loop induction variable and the scalar increment value. Using the latencies in Figure 4.1 and assuming a 1-cycle branch delay, write an expression for the total time for the software-pipelined loop to increment all elements of the array.

b. [20] <4.4> The original loop in the example may execute only one iteration. Write code for the complete software-pipelined loop that allows one, two, or as many iterations as needed for the array size. Then, for your final answer, schedule and possibly further transform your code so that it can execute with only two stalls, and show where those stalls occur. Assume the latencies in Figure 4.1, and use delayed branches that always execute the instruction in the 1-cycle branch delay slot.

4.11 [20] <4.4> Consider the loop that we software-pipelined on page 330. Suppose the latency of the `ADD.D` was 5 cycles. The software-pipelined loop now has a stall. Show how this loop can be written using both software pipelining and loop unrolling to eliminate any stalls. The loop should be unrolled as few times as possible (once is enough). Show the loop start-up and clean-up code.

✪ 4.12 [15] <4.4> Here is a simple code fragment:

```
for (i=2;i<=100;i+=2)
        a[i] = a[50*i+1];
```

To use the GCD test, this loop must first be "normalized"—written so that the index starts at 1 and increments by 1 on every iteration. Write a normalized version of the loop (change the indices as needed), then use the GCD test to see if there is a dependence.

4.13 [15] <4.4> Here is another loop:

```
for (i=2,i<=n/2;i+=2)
        a[i] = a[i] + a[i + n/2];
```

Normalize the loop. Does the GCD test detect a dependence? Is there a loop-carried, true dependence in this loop? Explain.

4.14 [25] <4.4> Show that if there is a true dependence between the two array elements A(a × i + b) and A(c × i + d), then GCD(c,a) divides (d – b).

4.15 [15] <4.5> It is common in scientific codes for array elements to be addressed based on the element values in an index array. Consider the following loop:

```
for (i=1,i<=n,i+=1)
        a[x[i]] = a[x[i]] + b[x[i]];
```

The array subscripts are not affine, so the GCD test cannot be used. However, the loop may still be parallel, so additional compiler tests may be valuable. What condition on the index array x[] will make the loop parallel? Be as general in your answer as you can. *Hint:* Think of the loop as adding the elements of one linked list to the corresponding elements of another linked list.

4.16 [15/15/15] <4.5> 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        0(R3),R1        ;store A
```

In the following assume a standard single-issue MIPS pipeline, branch resolution in the ID stage, delayed branches, and forwarding.

a. [15] <4.5> Assume conditional load instructions LWZ Rd,x(Rs1),Rs2 and LWNZ Rd,x(Rs1),Rs2 that do not load unless the value of Rs2 is zero or not zero, respectively. Compile the code using a conditional load and write it showing any stall cycles that would occur in the pipeline. Compare the clock cycles and register use to that of the original code fragment.

b. [15] <4.5> Boosting supports compiler speculation via a load instruction LW+ that includes a flag of the compiler prediction for the branch on which the load depends. If the prediction flag matches the branch resolution result, then the loaded value is written to the register. Compile the code using a boosted load and write it showing any stall cycles that would occur in the pipeline. Compare the clock cycles and register use to that of the original code fragment.

c. [15] <4.5> Compile the code using compiler-based speculation for both the then and else clauses and write it showing any stall cycles that would occur in the pipeline. Assume conditional move instructions CMVZ Rd,Rs1,Rs2 and CMVNZ Rd,Rs1,Rs2 that move the contents of register Rs1 to register Rd if Rs2 is equal to or not equal to zero, respectively. Compare the clock cycles and register use to that of the original code fragment.

✪ 4.17 [15] <4.5> Perform the same transformation (moving up the branch) as the example on page 342, but using only conditional move. Be careful that your loads, which are no longer control dependent, cannot raise an exception if they should not have been executed!

4.18 [15/15] <4.5> In this exercise we will investigate how predication affects the form and execution of pipelined instructions. Conditional execution of instructions is traditionally implemented with branches. For example, the MIPS instruction

```
DADD R1,R2,R3      ;R1=R2+R3
```

is unconditional. Its execution is made conditional on the value in register R8 by writing

```
            BNEZ R8,place  ;if(R8==0)
            DADD R1,R2,R3  ;then{ R1=R2+R3 }
place:
```

If predicated, however, the instruction form could be

```
            (R8) ADD R1,R2,R3 ;if(R8){ R1=R2+R3 }
place:
```

with control of execution of the ADD an integral part of the instruction itself, thus eliminating a control dependence between what was before two instructions. If R8 has been set to the value true, then the ADD computes the sum and places it in the result register; otherwise the ADD behaves like a NOP, computing no result and leaving R1 unchanged.

Assume a set of 1-bit predicate registers that are set by a compare instruction of the form

```
      (qp) CMP.NE pT,pF=R8,R0
```

This compare (written with mnemonic CMP) is itself predicated on the truth value of the qualifying predicate, qp. This example uses a not equal (.NE) comparison relation to match the code fragment above; other relations may be available. If qp is true, the CMP.NE sets the 1-bit predicate registers pT and pF such that pT=(R8!=R0) and pF!=(R8!=R0). That is, pT is assigned the truth value of the statement "R8 is not equal to R0," and pF is assigned the complement of the truth value of that same statement. If qp is false, the CMP.NE behaves as would a NOP instruction.

a. [15] <4.5> Using predicated instructions, write the following code fragment as a single basic block (assuming the SUB instruction is the only entry point for the code). You may assume that any compare instruction you use is not itself predicated.

```
        DSUB R1,R13,R14
        BNEZ R1,L1
        DADDI R2,R2,#1
        SD 0(R7),R2
        J L2
L1:     MUL.D F0,F0,F2
        ADD.D F0,F4
        S.D 0(R8),F0
L2:
```

b. [15] <4.5> What are all the dependences in the code given in part (a), and what are the dependences in the code for your answer to part (a)? How do they differ, and what is the advantage for performance of the predicated code?

4.19 [15/12/12] <4.5, 4.7> See the description of predicated instructions in the preceding exercise, then answer the following questions.

a. [15] <4.5, 4.7> Write the following code without branches. Use predicated MIPS instructions.

```
if (A>B) then { X=1;}
else {
             if (C<D) then { X=2;}
             else { X=3;} }
```

b. [12] <4.7> Where would an IA-64 compiler place stop(s) in the answer to part (a)?

c. [12] <4.7> What are the possible sequences of instruction bundle templates (see Figure 4.12) for the answer to part (a)? Assume that the first instruction begins a bundle.

✪ 4.20 [10] <4.5> Predicated instructions cannot eliminate a branch instruction when that branch is part of what kind of program structure?

4.21 [15]<4.7> A compiler for IA-64 has generated the following sequence of three instructions:

```
         L.D F0,0(R1)   ;F0=Mem[0+R1]
(p1)     DADD R1,R2,R3 ;if (p1) then R1=R2+R3
(p2)     DSUB R5,R1,R4 ;if (p2) then R5=R1-R4
```

where p1 and p2 are two predicate registers that are set earlier in the program. Assume that the three instructions are to form a bundle. What are the possible templates that the compiler could use for the bundle (see Figure 4.12), and under what circumstances would each template be chosen?

4.22 [20] <4.7> A compound conditional joins more than two values by Boolean operators, for example, `X&&Y&&Z`. If predicate-generating compare instructions can be ganged together to simultaneously update a single predicate register, then compound conditionals can be computed more rapidly. Consider a parallel computation of an && only compound condition (a conjunction term). If the predicate register were initialized to true, then simultaneous writes by only those compare instructions determining that their comparison should set the predicate to false (zero) is readily supportable in hardware. There will be no contention from trying to set the predicate simultaneously to conflicting values. A parallel not-equal compare to update `pT` might be written

```
(qp) CMP.NE.AND pT=Rx,R0
```

where `R0` always contains zero and `.AND` denotes that the write by this compare to predicate register `pT` will occur only if this compare finds `pT` is false. Initialize a predicate register to true, and then use the parallel compare instruction to transform the following code into a single block of predicated instructions and form it into as few bundles as possible, as in Figure 4.13 on page 355.

```
if (X && Y && Z) then { A=A+1;}
else { A=A+2;}
```

4.23 [10/20/10] <4.1, 4.5, 4.7, 5.10, 5.11> The example on page 342 uses a speculative load instruction to move a load above its guarding branch instruction. Consider the following code:

```
          instr. 1                        ;arbitrary instruction
          instr. 2                        ;next instruction in block
          . . .                           ;intervening instructions
          BEQZ       R1, null             ;check for null pointer
          L.D        F2,0(R1)             ;load using pointer
          ADD.D      F4,F0,F2             ;dependent ADD.D
          . . .
null:     . . .                           ;handle null pointer
```

a. [10] <4.5> Write the above code using a speculative load (sL.D) and a speculation check instruction (SPECCK) to preserve exception behavior. Where should the L.D move to best hide its potentially long latency?

b. [20] <4.1, 4.5, 4.7> Assume a speculation check instruction that branches to recovery code. Write the above code speculating on both the load and the dependent add. Use a speculative load, a nonspeculative add, a check instruction, and the block of recovery code. How should the speculated load and add be scheduled with respect to each other?

c. [10] <5.10, 5.11> What type(s) of load exceptions could the SPECCK protect for in part (a)? What type of load exception will trigger the recovery code in part (b)?

✪ 4.24 [15] <4.7> An advanced load address table (ALAT) holds the effective address of a load that has been moved before a preceding store, which may or may not have the same or overlapping effective address. For which processor, the one of Figure 4.1 or the IA-64 described in Figure 4.15, is an ALAT most beneficial and why?

4.25 [15] <4.8> Why might speculation and predication be of less value in the embedded computer marketplace than in the server or desktop arena? What are the market niches where they will be least valued?

4.26 [20] <4.1, 4.4, 4.5> Branches are the target of considerable effort for dynamically scheduled processors. For statically scheduled processors, loop unrolling, trace scheduling, superblocks, and predication all attempt to reduce the negative effects of branches on performance. Construct a comparison of these four techniques. For each technique include a description of its best-suited branch characteristics, suitable program structures, needed hardware support, complexity of compiler support, effect on code size, effect on fetching, and other meaningful distinguishing features.

4.27 [Discussion] <3.2–3.7, 4.1–4.5> Dynamic instruction scheduling requires a considerable investment in hardware. In return, this capability allows the hardware to run programs that could not be run at full speed with only compile time, static scheduling. What trade-offs should be taken into account in trying to decide between a dynamically and a statically scheduled implementation? What situations in either hardware technology or program characteristics are likely to favor one approach or the other? Most speculative schemes rely on dynamic scheduling; how does speculation affect the arguments in favor of dynamic scheduling? Many static schemes incorporate predication; how do branch behavior and program structure affect the arguments in favor of predication?

4.28 [Discussion] <3.2, 3.3, 4.5> Consider combining the static and dynamic ILP techniques of predicated instructions and the Tomasulo algorithm. How might predicated instructions be handled in each of the three steps of the Tomasulo algorithm (see Section 3.2)? For each approach that you can devise, clearly define how it is implemented, discuss its performance potential, list what additional hardware support is necessary if any, tell whether the approach involves speculative execution or not, and identify whether dependences on the predicate behave as data or control dependences. Do you think the two techniques work together well?

4.29 [Discussion] <4.3–4.5> Discuss the advantages and disadvantages of a superscalar implementation and a VLIW approach in the context of MIPS. What levels of ILP favor each approach? What other concerns would you consider in choosing which type of processor to build? How does speculation affect the results?

4.30 [Discussion] <3, 4> Investigate the delivered clock speeds for various processors using primarily hardware techniques for exploiting ILP and for processors focused on software techniques for exploiting ILP. Try to determine the reasons for any differences in clock speeds. Examine available benchmark results to see how they do or do not correlate closely with clock speed.