For example, the IBM RS/6000 Power 2 model 900 can issue up to six instructions per clock cycle, and its data cache can supply two 128-bit accesses per clock cycle. The RS/6000 does this by making the instruction cache and data cache wide and by making two reads to the data cache each clock cycle, certainly likely to be the critical path in the 71.5-MHz machine.

Speculative Execution and the Memory System

Inherent in CPUs that support speculative execution or conditional instructions is the possibility of generating invalid addresses that would not occur without speculative execution. Not only would this be incorrect behavior if exceptions were taken, the benefits of speculative execution would be swamped by false exception overhead. Hence the memory system must identify speculatively executed instructions and conditionally executed instructions and suppress the corresponding exception.

By similar reasoning, we cannot allow such instructions to cause the cache to stall on a miss, for again unnecessary stalls could overwhelm the benefits of speculation. Hence these CPUs must be matched with nonblocking caches (see page 414).

Compiler Optimization: Instruction-Level Parallelism versus Reducing Cache Misses

Sometimes the compiler must choose between improving instruction-level parallelism and improving cache performance. For example, the code below,

```
for (i = 0; i < 512; i = i+1)
      for (j = 1; j < 512; j = j+1)
            x[i][j] = 2 * x[i][j-1];
```

accesses the data in the order they are stored, thereby minimizing cache misses. Unfortunately, the dependency limits parallel execution. Unrolling the loop shows this dependency:

```
for (i = 0; i < 512; i = i+1)
      for (j = 1; j < 512; j = j+4){
            x[i][j]   = 2 * x[i][j-1];
            x[i][j+1] = 2 * x[i][j];
            x[i][j+2] = 2 * x[i][j+1];
            x[i][j+3] = 2 * x[i][j+2];
            };
```

For example, the IBM RS/6000 Power 2 model 900 can issue up to six instructions per clock cycle, and its data cache can supply two 128-bit accesses per clock cycle. The RS/6000 does this by making the instruction cache and data cache wide and by making two reads to the data cache each clock cycle, certainly likely to be the critical path in the 71.5-MHz machine.

Speculative Execution and the Memory System

Inherent in CPUs that support speculative execution or conditional instructions is the possibility of generating invalid addresses that would not occur without speculative execution. Not only would this be incorrect behavior if exceptions were taken, the benefits of speculative execution would be swamped by false exception overhead. Hence the memory system must identify speculatively executed instructions and conditionally executed instructions and suppress the corresponding exception.

By similar reasoning, we cannot allow such instructions to cause the cache to stall on a miss, for again unnecessary stalls could overwhelm the benefits of speculation. Hence these CPUs must be matched with nonblocking caches (see page 414).

Compiler Optimization: Instruction-Level Parallelism versus Reducing Cache Misses

Sometimes the compiler must choose between improving instruction-level parallelism and improving cache performance. For example, the code below,

```
for (i = 0; i < 512; i = i+1)
      for (j = 1; j < 512; j = j+1)
            x[i][j] = 2 * x[i][j-1];
```

accesses the data in the order they are stored, thereby minimizing cache misses. Unfortunately, the dependency limits parallel execution. Unrolling the loop shows this dependency:

```
for (i = 0; i < 512; i = i+1)
      for (j = 1; j < 512; j = j+4){
            x[i][j]   = 2 * x[i][j-1];
            x[i][j+1] = 2 * x[i][j];
            x[i][j+2] = 2 * x[i][j+1];
            x[i][j+3] = 2 * x[i][j+2];
            };
```


Each of the last three statements has a RAW dependency on the prior statement. We can improve parallelism by interchanging the two loops:

```
for (j = 1; j < 512; j = j+1)
      for (i = 0; i < 512; i = i+1)
            x[i][j] = 2 * x[i][j-1];
```

Unrolling the loop shows this parallelism:

```
for (j = 1; j < 512; j = j+1)
      for (i = 0; i < 512; i = i+4) {
            x[i][j]   = 2 * x[i][j-1];
            x[i+1][j] = 2 * x[i+1][j-1];
            x[i+2][j] = 2 * x[i+2][j-1];
            x[i+3][j] = 2 * x[i+3][j-1];
            };
```

Now all four statements in the loop are independent! Alas, increasing parallelism leads to accesses that hop through memory, reducing spatial locality and cache hit rates.

I/O and Consistency of Cached Data

Because of caches, data can be found in memory and in the cache. As long as the CPU is the sole device changing or reading the data and the cache stands between the CPU and memory, there is little danger in the CPU seeing the old or *stale* copy. I/O devices give the opportunity for other devices to cause copies to be inconsistent or for other devices to read the stale copies. Figure 5.46 illustrates the problem, generally referred to as the *cache-coherency* problem.

The question is this: Where does the I/O occur in the computer—between the I/O device and the cache or between the I/O device and main memory? If input puts data into the cache and output reads data from the cache, both I/O and the CPU see the same data, and the problem is solved. The difficulty in this approach is that it interferes with the CPU. I/O competing with the CPU for cache access will cause the CPU to stall for I/O. Input will also interfere with the cache by displacing some information with the new data that is unlikely to be accessed by the CPU soon. For example, on a page fault the CPU may need to access a few words in a page, but a program is not likely to access every word of the page if it were loaded into the cache. Given the integration of caches onto the same integrated circuit, it is also difficult for that interface to be visible.

The goal for the I/O system in a computer with a cache is to prevent the stale-data problem while interfering with the CPU as little as possible. Many systems, therefore, prefer that I/O occur directly to main memory, with main memory