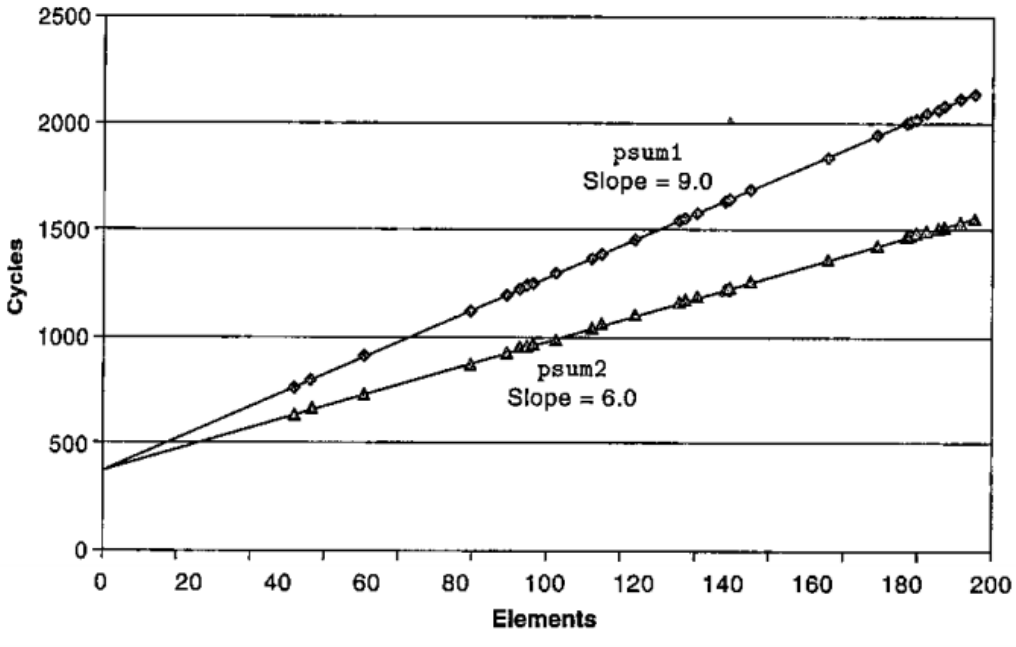
Code Optimization

The first step in optimizing. a program is to eliminate unnecessary work, making the code perform its intended task .as efficiently as possible. This includes eliminating unnecessary function calls, conditional tests, and memory references. These optimizations do not depend on any specific properties of the target machine. The second step is exploiting the capability of processors to provide instruction-level parallelism, executing multiple instructions simultaneously.

The cycles per element, abbreviated CPE, to express program performance in a way that can guide us in improving the code. The sequencing of activities by a processor is controlled by a clock providing. Clock rate : Measured in Megahertz or Gigahertz ; Function of stage partitioning and circuit design. All in all, the basic principles of Stage design: the workload of each stage as little as possible. The size of Stage is as small as possible.

For function of pipeline design and benchmark programs.

Q : how frequently are branches mispredicted?



(CPE is slope of line)

For a set of data points (x1, y1), ... , (Xn , Yn), we often try to draw a line that best approximates the X-Y trend represented by these data. With a least squares fit, we look for a line of the form y = mx + b that minimizes the following error measure : *E(m, b) ~ L (mx;.,+*

b ~ y,)^2. An algorithm for computing *m* and *b* can be derived by finding the derivatives of E (m, b) with respect to m and b and setting them to 0.

To demonstrate how an abstract program can be systematically transformed into more efficient code. In our evaluation, we measured the performance of our code for integer (C int and long), and floating-point (C float and double) data.

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

Initial implementation of combining operation. Using different declarations *of* identity element IDENT and combining operation OP, we can measure the routine for different operations.

As a starting point, the following table shows CPE measurements for combine1 running on our reference machine, with different combinations of operation (addition or multiplication) and data type (long integer and double precision floating point). Our experiments with many different programs showed that operations on 32-bit and 64-bit integers have identical performance, with the exception of code involving division operations. Similarly, we found identical performance for programs operating on single- or double-precision floating-point data. In our tables, we will therefore show only separate results for integer data and for floating-point data.

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14
Combine1 –O3	4.5	4.5	6	7.8

The combine2 calls vec_length at the beginning and assigns the result to a local variable length. This transformation has noticeable effect on the overall performance for some data types and

operations, and minimal or even none for others. In any case, this transformation is required to eliminate inefficiencies that would become bottlenecks as we attempt further optimizations.

			Integer		Floating point	
Function	Page	Method	+	*	+	*
combine1	507	Abstract -01	10.12	10.12	10.17	11.14
combine2	509	Move vec_length	7.02	9.03	9.02	11.03

```
/* Move call to vec_length out of loop */
void combine2(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);

    *dest = IDENT;
    for (i = 0; i < length; i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

We can therefore move the computation to an earlier section of the code that does not get evaluated as often. In this case, we moved the call to vec_length from within the loop to just before the loop. Optimizing compilers attempt to perform code motion. Unfortunately, as discussed previously, they are typically very cautious about making transformations that change where or how many times a procedure is called. They cannot reliably detect whether or not a function will have side effects, and 5o they assume that it might.

As we have known, procedure calls can incur overhead and also block most forms of program optimization. We can see in the code for combine2 (Figure 5.6) that get_ vec_element is called on every loop iteration to retrieve the next yector element. This function checks the vector index i against the loop bounds with every vector reference, a clear source of inefficiency.

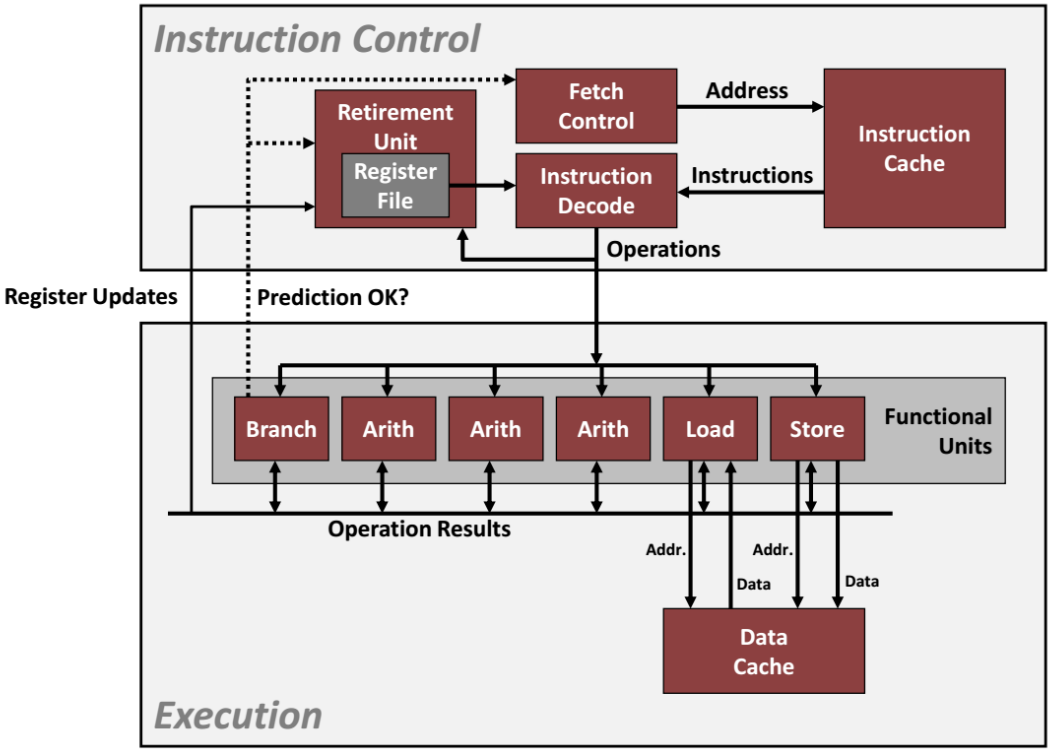
So Eliminating function calls within the loop. The resulting code does not show a performance gain, but it enables additional optimizations.

Function	Page	Method	Integer		Floating point	
			+	*	+	*
`combine2`	509	Move `vec_length`	7.02	9.03	9.02	11.03
`combine3`	513	Direct data access	7.17	9.02	9.02	11.03

By Eliminating Unneeded Memory References, We can eliminate this needless reading. Accumulating result in temporary. Holding the accumulated value in local variable ace (short for "accumulator") eliminates' the need to retrieve it from memory and write back the updated value on every loop iteration. We see a significant improvement in program performance, as shown in the following table:

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O1	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01

We can see a block diagram of an out-of-order processor. The instruction control unit is responsible for reading instructions from memory and generating a sequence of primitive operations. The execution unit then performs the operations and indicates whether the branches were correctly predicted. Up to this point, we have applied optimizations that did not rely on any features of the target machine. They simply reduced the overhead of procedure calls and eliminated some of the critical "optimization blockers" that cause difficulties for optimizing compilers. As we seek to push the performance further, we must consider optimizations that exploit the *microarchitecture* of the processor-that is, the underlying system design by which a processor executes instructions. Getting every last bit of performance requires a detailed analysis of the program as well as code generation tuned for the target processor. Nonetheless, we can apply some basic optimizations that will yield an overall performance improvement on a large class of processors. The detailed performance results we report here may not hold for other machines, but the general principles of operation and optimization apply to a wide variety of machines.



Most modern CPUs are superscalar. Benefit: without programming effort, superscalar processor can take advantage of the instruction level parallelism that most programs have.

Figure 5.11 shows a very simplified view of a modern microprocessor. Our hypothetical processor design is based loosely on the structure of recent Intel processors. These processors are described in the industry as being superscalar, which means they can perform multiple operations on every clock cycle and out of order, meaning that the order in which instructions execute need not correspond to their ordering in the machine-level program. The overall design has two main parts: the instruction control unit (ICU), which is responsible for reading a sequence of instructions from memory and generating from these a set of primitive operations to perform on program data, and the execution unit (EU), which then executes these operations. Compared to the simple in-order pipeline we studied in Chapter 4, out-of-order processors require far greater and more complex hardware, but they are better at achieving higher degrees of instruction-level parallelism.

The ICU reads the instructions from an instruction cache-a special high speed memory containing the most recently accessed instructions. In .general, the ICU fetches well ahead of the currently executing instructions, so that it has enough time to decode these and send operations down to the EU. One problem, however, is that when a program hits a branch, there are two possible directions the program might go. The branch can be taken, with control passing to

the branch target. Alternatively, the branch can be not taken, with control passing to the next instruction in the instruction sequence. Modern processors employ a technique known as branch prediction, in which they guess whether or not a branch will be taken and also predict the target address for the branch. Using a technique known as speculative execution, the processor begins fetching and decoding instructions at where it predicts the branch will go, and even begins executing these operations before it has been determined whether or not the branch prediction was correct. If it later determines that the branch was predicted incorrectly, it resets the state to that at the branch point and begins fetching and executing instructions in the other direction. The block labeled "Fetch control" incorporates branch prediction to perform the task of determining which instructions to fetch.

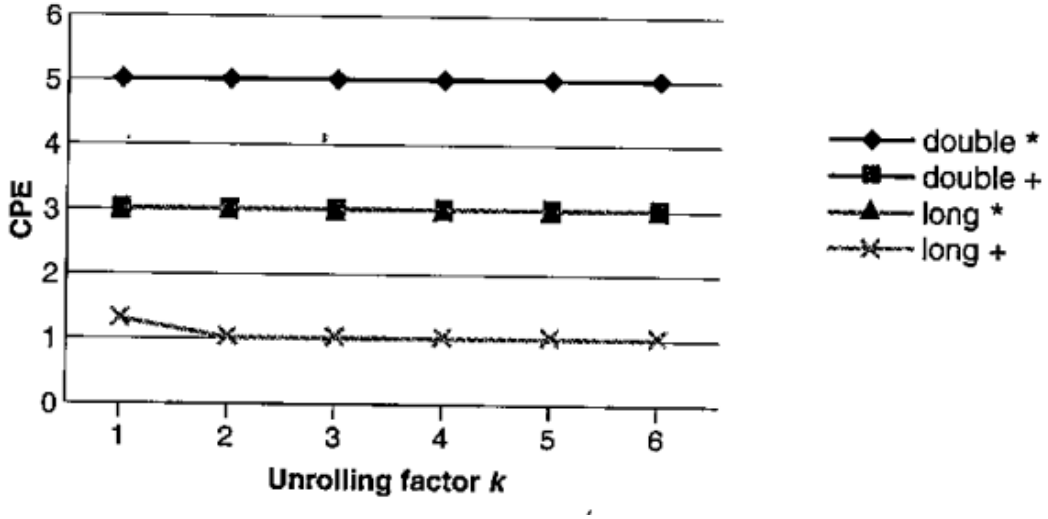
Intel: Haswell CPU (8 Total Functional Units)
Multiple instructions can execute in parallel
2 load, with address computation
1 store, with address computation
4 integer
2 FP multiply
1 FP add
1 FP divide
Some instructions take > 1 cycle, but can be pipelined

Instruction	*Latency*	*Cycles/Issue*
Load / Store	4	1
Integer Multiply	3	1
Integer/Long Divide	**3-30**	**3-30**
Single/Double FP Multiply	5	1
Single/Double FP Add	3	1
Single/Double FP Divide	**3-15**	**3-15**

Loop unrolling is a program transformation that reduces the number of iterations for a loop by increasing the number of elements. Computed on each iteration. Loop unrolling can improve performance in two ways. First, it reduces the number of operations that do not contribute directly to the program result, such as loop indexing and conditional branching. Second, it exposes ways in which we can further transform the code to reduce the number of operations in the critical paths of the overall computation.

We see that the CPE for integer addition improves, achieving the latency bound of 1.00. This result can be attributed to the benefits of reducing loop overhead operations. By reducing the number of overhead operations relative to the number of additions required to

compute the vector sum, we can reach the point where the 1-cycle latency of integer addition becomes the performance limiting factor. On the other hand, none of the other cases improve-they are already at their latency bounds. Figure shows CPE measurements when unrolling the loop by up to a factor of 10. We see that the trends we observed for unrolling by 2 and 3 continue-none go below their latency bounds.



Although we have only considered a limited set of applications, we can draw important lessons on how to write efficient code. We 'have described a number of basic strategies for optimizing program performance:

1. High-level design. Choose appropr1'1te algorithms and data structures for the problem at hand. Be especially vigilant to avoid algorithms or coding techniques that yield asymptotically poor performance.
2. Basic coding principles. Avoid optimization blockers so that a compiler can generate efficient code. ; Eliminate excessive function calls. Move computations out of loops when possible. Consider selective compromises of program modularity to gain greater efficiency; Eliminate unnecessary memory references. Introduce temporary variables to hold intermediate results. Store a result in an array or global variable only when the final value has been computed.
3. Low-level optimizations. Structure code to take advantage of the hardware capabilities: Unroll loops to reduce overhead and to enable further optimization& Find ways to increase instruction-level parallelism by techniques such as multiple accumulators and reassociation ; Rewrite conditional operations in a functional style to enable compilation via conditional data transfers.

Effect of Separate Accumulators

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
Latency Bound	*1.00*	*3.00*	*3.00*	*5.00*
Throughput Bound	*0.50*	*1.00*	*1.00*	*0.50*

Using Vector Instructions

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Scalar Best	0.54	1.01	1.01	0.52
Vector Best	0.06	0.24	0.25	0.16
Latency Bound	*0.50*	*3.00*	*3.00*	*5.00*
Throughput Bound	*0.50*	*1.00*	*1.00*	*0.50*
Vec Throughput Bound	*0.06*	*0.12*	*0.25*	*0.12*

Summary (How to Getting High Performance)

1. Good compiler and flags
2. Don’t do anything stupid
   - Watch out for hidden algorithmic inefficiencies
   - Write compiler-friendly code

   Watch out for optimization blockers ; procedure calls & memory references ; Look carefully at innermost loops (where most work is done)
3. Tune code for machine
   - Exploit instruction-level parallelism
   - Avoid unpredictable branches
   - Make code cache friendly (Covered later in course)