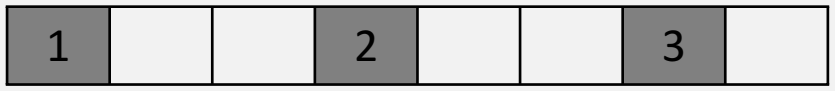
#8: Compiler Optimizations

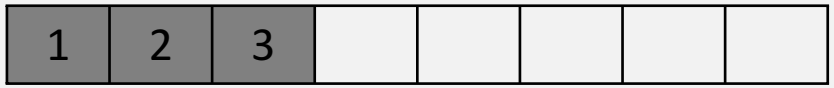
Loop Interchange

Reordering nested loops to access memory in sequential order, improving spatial locality and maximizing use of cache blocks.

```
/* Before */
for (j = 0; j < 100; j++)
  for (i = 0; i < 5000; i++)
    x[i][j] = 2 * x[i][j];
```




```
/* After */
for (i = 0; i < 5000; i++)
  for (j = 0; j < 100; j++)
    x[i][j] = 2 * x[i][j];
```



#8: Compiler Optimizations (con't)

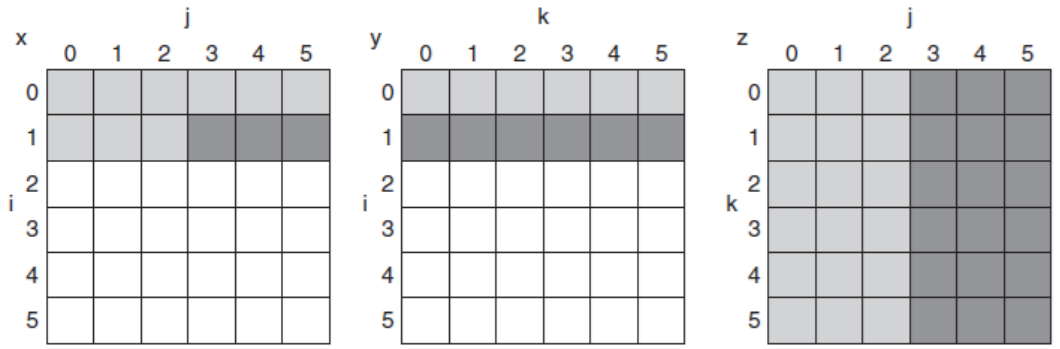
Blocking

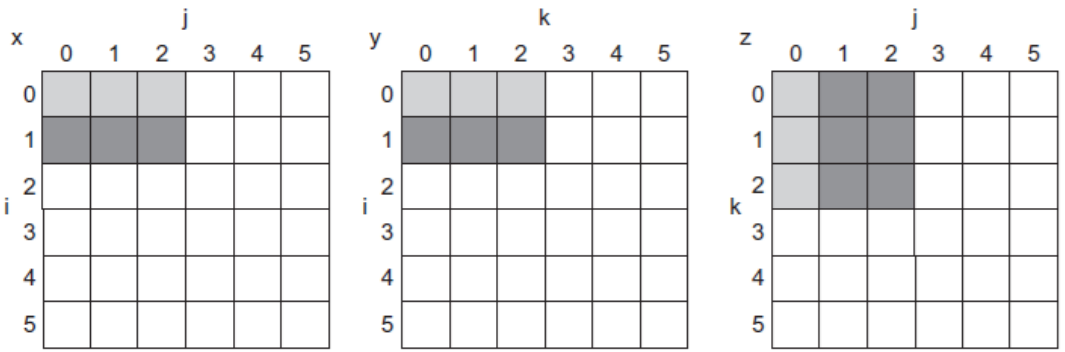
- This optimization improves temporal locality to reduce misses.
  - We are again dealing with multiple arrays, with some arrays accessed by rows and some by columns. Storing the arrays row by row (row major order) or column by column (column major order) does not solve the problem because both rows and columns are used in every loop iteration.
- Instead of operating on entire rows or columns of an array, blocked algorithms operate on submatrices or blocks.

Blocking

```
/* Before */
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++) {
    r = 0;
    for (k = 0; k < N; k++)
      r = r + y[i][k]*z[k][j];
    x[i][j] = r;
  };
```

```
/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i++)
  for (j = jj; j < min(jj+B,N); j++) {
    r = 0;
    for (k = kk; k < min(kk+B,N); k++)
      r = r + y[i][k]*z[k][j];
    x[i][j] = x[i][j] + r;
  };
```





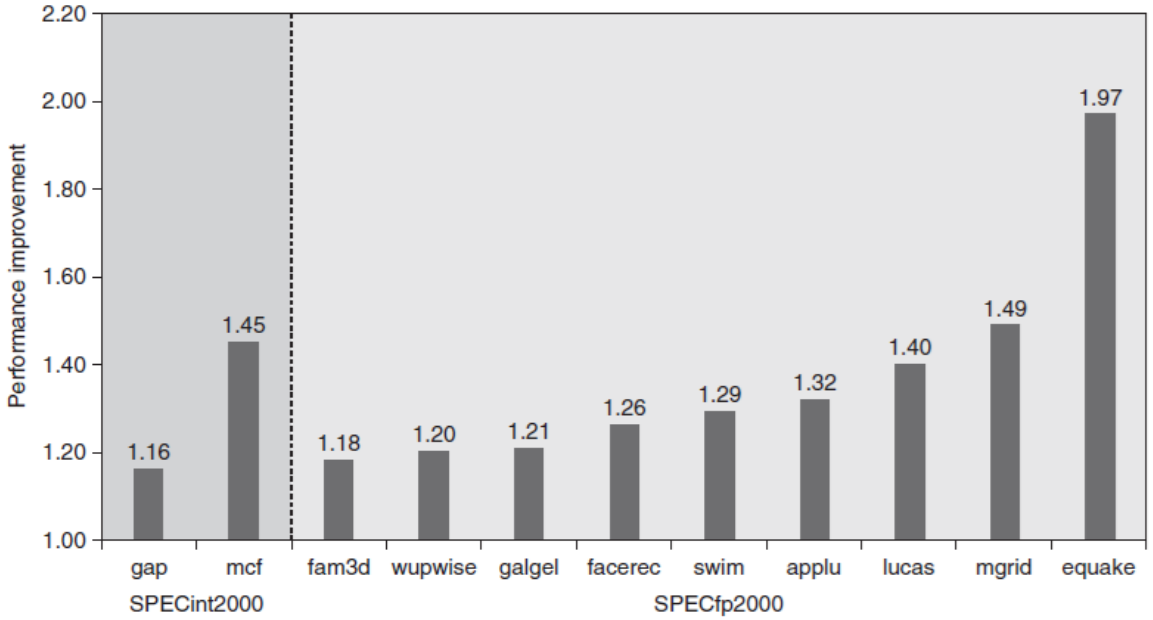
#9: Hardware Prefetching

The Technique

Prefetch items before the processor requests them, either into caches or external buffers.

- **Instruction prefetch:** Fetch the requested block and the next consecutive block
- **Data prefetch:** Similar approach with stream buffers for data

Intel Core i7 supports hardware prefetching into both L1 and L2, typically prefetching the next cache line.



Speedup due to hardware prefetching on Intel Pentium 4 with hardware prefetching

Prefetching relies on utilizing otherwise unused memory bandwidth, but can lower performance if it interferes with demand misses. It has minimal impact on power when working well, but can be very negative when prefetched data isn't used.

#10: Compiler-Controlled Prefetching

- Types of Prefetch
  - **Register prefetch**: Load value into a register
  - **Cache prefetch**: Load data only into the cache
  - Either can be faulting or non-faulting (doesn't incur a page fault)
    - Most modern processors offer non-faulting cache prefetches.
- Implementation Considerations
  - Requires non-blocking caches to be effective
  - Incurs instruction overhead
  - Must focus on likely cache misses to avoid unnecessary prefetches
  - Most effective with loops and predictable access patterns
- Prefetching can provide 4-31% improvement in programs with recursive data structures.

Example

- Let's assume we have an 8 KB direct-mapped data cache with 16-byte blocks, and it is a write-back cache that does write allocate.
- The elements of `a` and `b` are 8 bytes long since they are double-precision floating-point arrays.
- There are 3 rows and 100 columns for `a` and 101 rows and 3 columns for `b`.

```
for (i = 0; i < 3; i++)
  for (j = 0; j < 100; j++)
    a[i][j] = b[j][0] * b[j+1][0];
```

- Elements of `a` are written in the order that they are stored in memory, so `a` will benefit from spatial locality
  - The even values of `j` will miss and the odd values will hit. Since a has 3 rows and 100 columns, its accesses will lead to 3 × (100/2), or 150 misses.
- The array `b` does not benefit from spatial locality since the accesses are not in the order it is stored. The array `b` does benefit twice from temporal locality
  - The same elements are accessed for each iteration of `i`, and each iteration of `j` uses the same value of `b` as the last iteration.
  - Misses due to `b` will be for `b[j+1][0]` accesses when `i = 0`, and also the first access to `b[j][0]` when `j = 0`. Since `j` goes from 0 to 99 when `i = 0`, accesses to `b` lead to 100 + 1, or 101 misses.

Example - Prefetching

```
for (j = 0; j < 100; j++) {
  prefetch(b[j+7][0]);
  prefetch(a[0][j+7]);
  a[0][j] = b[j][0] * b[j+1][0];
};
for (i = 1; i < 3; i++)
  for (j = 0; j < 100; j++) {
    prefetch(a[i][j+7]);
    a[i][j] = b[j][0] * b[j+1][0];
  }
```

- Let's assume that the miss penalty is so large we need to start prefetching at least, say, seven iterations in advance.
- Total of 19 non-prefetched misses.
  - 7 misses for elements `b[0][0], b[1][0], . . . , b[6][0]` in the first loop
  - 4 misses ([7/2]) for elements `a[0][0], a[0][1], . . . , a[0][6]` in the first loop (spatial locality reduces misses to 1 per 16-byte cache block)
  - 4 misses ([7/2]) for elements `a[1][0], a[1][1], . . . , a[1][6]` in the second loop
  - 4 misses ([7/2]) for elements `a[2][0], a[2][1], . . . , a[2][6]` in the second loop
- The cost of avoiding 232 cache misses is executing 400 prefetch instructions, likely a good trade-off.

Cache Optimization Summary

Technique	Hit time	Band-width	Miss penalty	Miss rate	Power consumption	Hardware cost/ complexity	Comment
Small and simple caches	+			–	+	0	Trivial; widely used
Way-predicting caches	+				+	1	Used in Pentium 4
Pipelined cache access	–	+				1	Widely used
Nonblocking caches		+	+			3	Widely used
Banked caches		+			+	1	Used in L2 of both i7 and Cortex-A8
Critical word first and early restart			+			2	Widely used
Merging write buffer			+			1	Widely used with write through
Compiler techniques to reduce cache misses				+		0	Software is a challenge, but many compilers handle common linear algebra calculations
Hardware prefetching of instructions and data			+	+	–	2 instr., 3 data	Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware.
Compiler-controlled prefetching			+	+		3	Needs nonblocking cache; possible instruction overhead; in many CPUs