Advance Cache optimization

I would review some of the optimizations to improve cache performance based on metrics like hit time, miss rate, miss penalty, cache bandwidth and power consumption. Most of the techniques described below would be based on hardware while some will rely on software/programming techniques. I would start by reviewing in depth about the software technique, mainly Compiler Optimizations and then go on to review other hardware-based optimizations.

Compiler Optimizations

The technique relies on optimization of software rather than hardware. The two techniques that we are going to discuss are Loop Interchange and Blocking.

Loop Interchange

Some programs have nested loops that access data in memory in nonsequential order. Simply exchanging the nesting of the loops can make the code access the data in the order in which they are stored. Since the arrays are laid out in row-major fashion accessing them in different ways can affect miss rate greatly.

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

The original code would skip through memory in strides of 100 words, while the revised version accesses all the words in one cache block before going to the next block. This optimization improves cache performance without affecting the number of instructions executed.

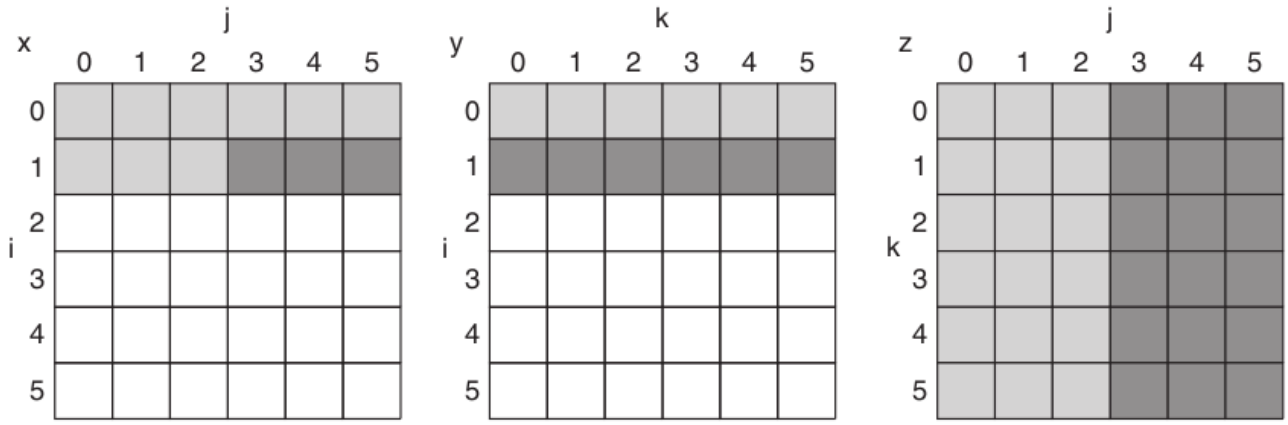
Blocking

This optimization improves temporal locality to reduce misses. When we need to deal with multiple arrays, with some arrays accessed by rows and some by columns,

storing in a specific way doesn't solve the problem. Such orthogonal accesses mean that transformations such as loop interchange still leave plenty of room for improvement. Consider the example code given below.

```
/* 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;
  }
```

The number of capacity misses clearly depends on N and the size of the cache. If it can hold all three N-by-N matrices, then all is well, provided there are no cache conflicts. If the cache can hold one N-by-N matrix and one row of N, then at least the ith row of y and the array z may stay in the cache. Less than that and misses may occur for both x and z. In the worst case, there would be $2N^3+N^2$ memory words accessed for N^3 operations.

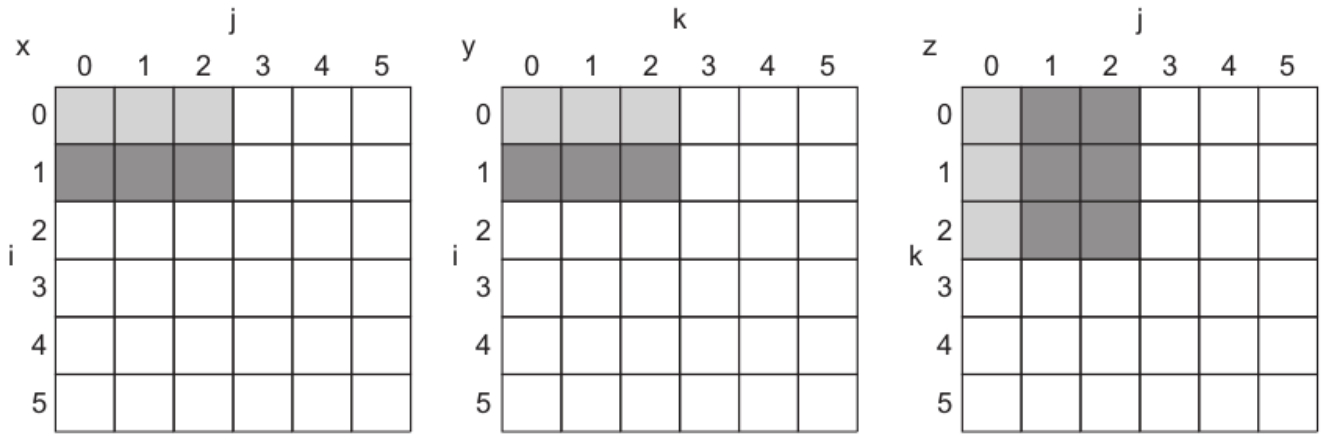


Access Pattern: Naive Implementation for matrix multiplication

Instead of operating on entire rows or columns of an array, blocked algorithms operate on submatrices or blocks. The goal is to maximize accesses to the data loaded into the cache before the data are replaced. To ensure that the elements being accessed can fit in the cache, the original code is changed to compute on a submatrix of size B by B. Two inner loops now compute in steps of size B rather than the full length of x and z. B is called the blocking factor.

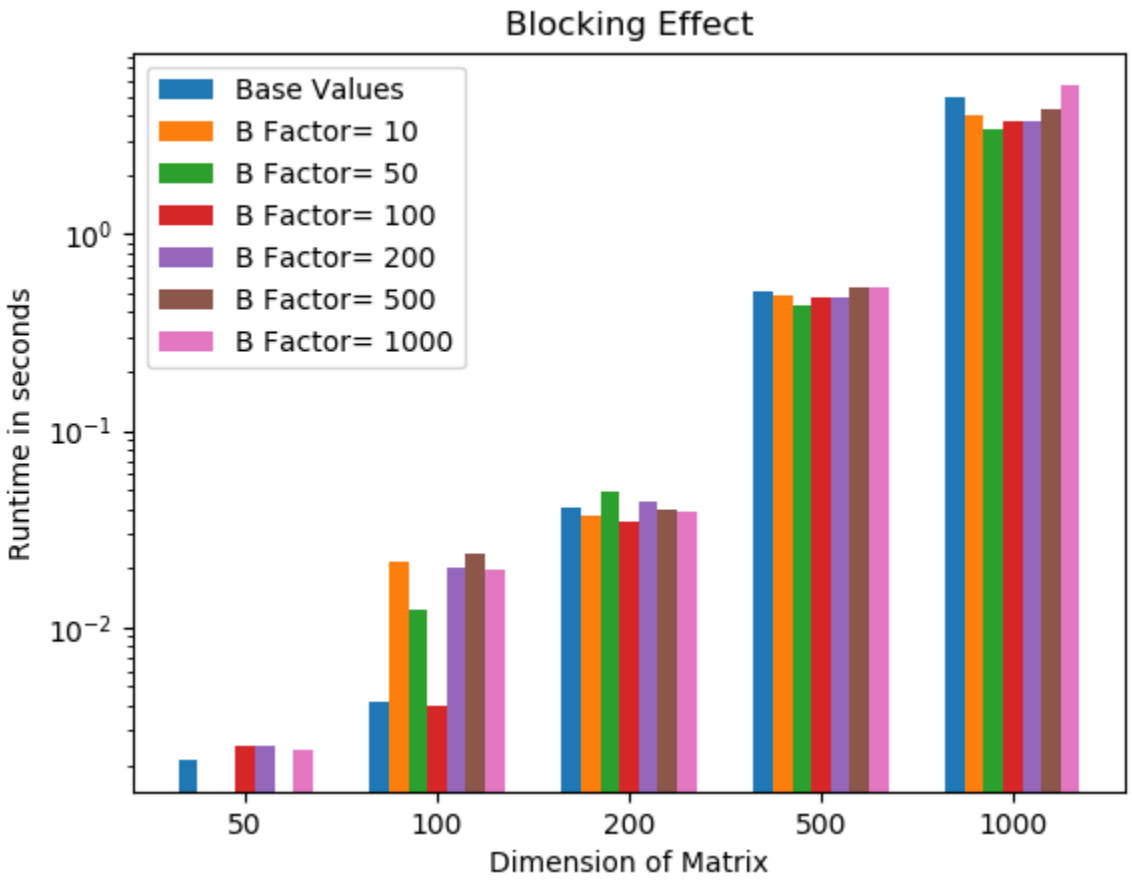
```
/* After */
for (jj = 0; jj < N; jj = jj + B)
  for (kk = 0; kk < N; kk = kk + B)
    for (i = 0; i < N; i++)
      for (j = jj; j < min(jj+B, N); j++)
      {
        r = 0;
        for (k = kk; k < min(kk+B, N); k++)
          r = r + y[i][k]*z[k][j];
        x[i][j] = r + x[i][j];
      }
```

Looking only at capacity misses, the total number of memory words accessed is $2N^3/B+ N^2$. This total is an improvement by about a factor of B. Hence, blocking exploits a combination of spatial and temporal locality, since y benefits from the spatial locality and z benefits from the temporal locality.



Access Patterns: Matrix multiplication using blocking algorithm

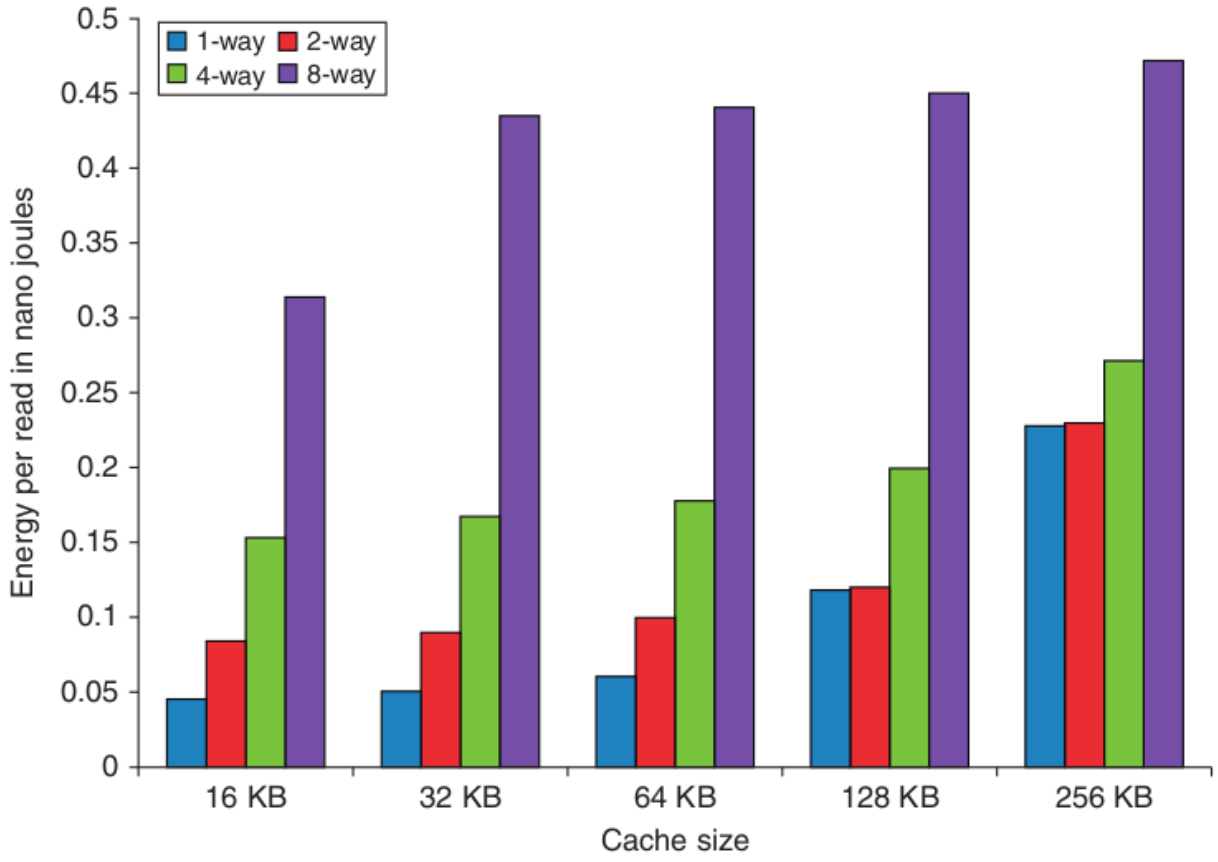
I have summarised the results of the above example for different values of N and B in the graph below, which illustrates the better cache utilisation in case of blocking algorithm as compared to simple matrix multiplication algorithm.



Graphical Representation: Blocking Effect

Small and Simple First-Level Caches

The cache hit is a three-step process of addressing tag memory, comparing the read tag value to the address and setting the multiplexor to choose the correct data item for associative caches. Since direct-mapped caches can overlap the tag check with the transmission of data, it reduces the hit time. Also, lower levels of associativity reduce power consumption since a fewer number of cache lines needs to be accessed.



Energy consumption per read

Way Prediction

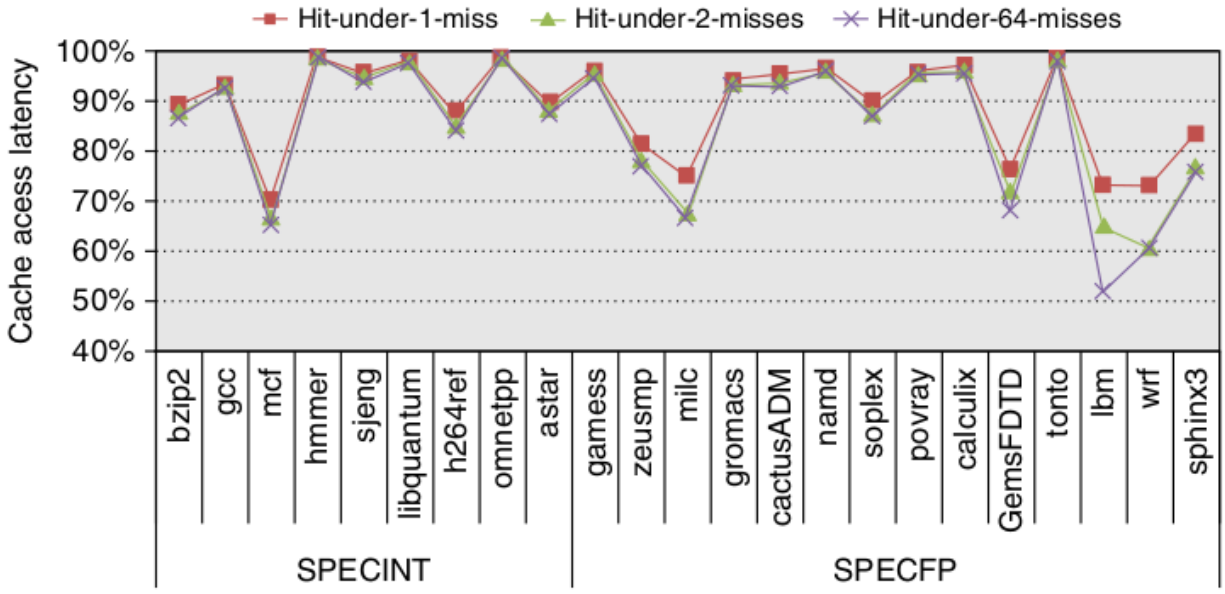
In way prediction, extra bits are kept in the cache to predict
the way, or block within the set of the next cache access. This prediction means the multiplexor is set early to select the desired block, and only a single tag comparison is performed that clock cycle in parallel with reading the cache data. A miss results in checking the other blocks for matches in the next clock cycle. Added to each block of a cache are block predictor bits. The bits select which of the blocks to try on the next cache access. If the predictor is correct, the cache access latency is the fast hit time. If not, it tries the other block, changes the way predictor, and has a latency of one extra clock cycle. This approach reduces conflict misses and yet maintains the hit speed of
direct-mapped cache.

Pipelined Cache Access

This optimization is simply to pipeline cache access so that the effective latency of a first-level cache hit can be multiple clock cycles, giving fast clock cycle time and high bandwidth but slow hits. This change increases the number of pipeline stages, leading to a greater penalty on mispredicted branches and more clock cycles between issuing the load and using the data, but it does make it easier to incorporate high degrees of associativity.

Non Blocking Caches

For pipelined computers that allow out-of-order execution, the processor need not stall on a data cache miss. A *nonblocking cache* or *lockup-free cache* escalates the potential benefits of such a scheme by allowing the data cache to continue to supply cache hits during a miss. This optimization reduces the effective miss penalty by being helpful during a miss instead of ignoring the requests of the processor.



The effectiveness of a nonblocking cache is evaluated by allowing 1, 2, or 64 hits under a cache miss with 9 SPECINT (on the left) and 9 SPECFP (on the right)

Multibanked Caches

According to this concept, we can divide cache into independent banks that can support simultaneous accesses rather than treating cache as a single monolithic block. Banking works best when the accesses naturally spread themselves across the banks, so the mapping of addresses to banks affects the behaviour of the

memory system. A simple mapping that works well is to spread the addresses of the block sequentially across the banks, called sequential interleaving.

Critical Word First and Early Restart

This technique is based on the observation that the processor normally needs just one word of the block at a time. With this approach, the processor doesn't wait for the full block to be loaded before sending the requested word and starting the processor. These techniques only benefit designs with large cache blocks. Two specific strategies are *Critical word first* and *Early restart.*

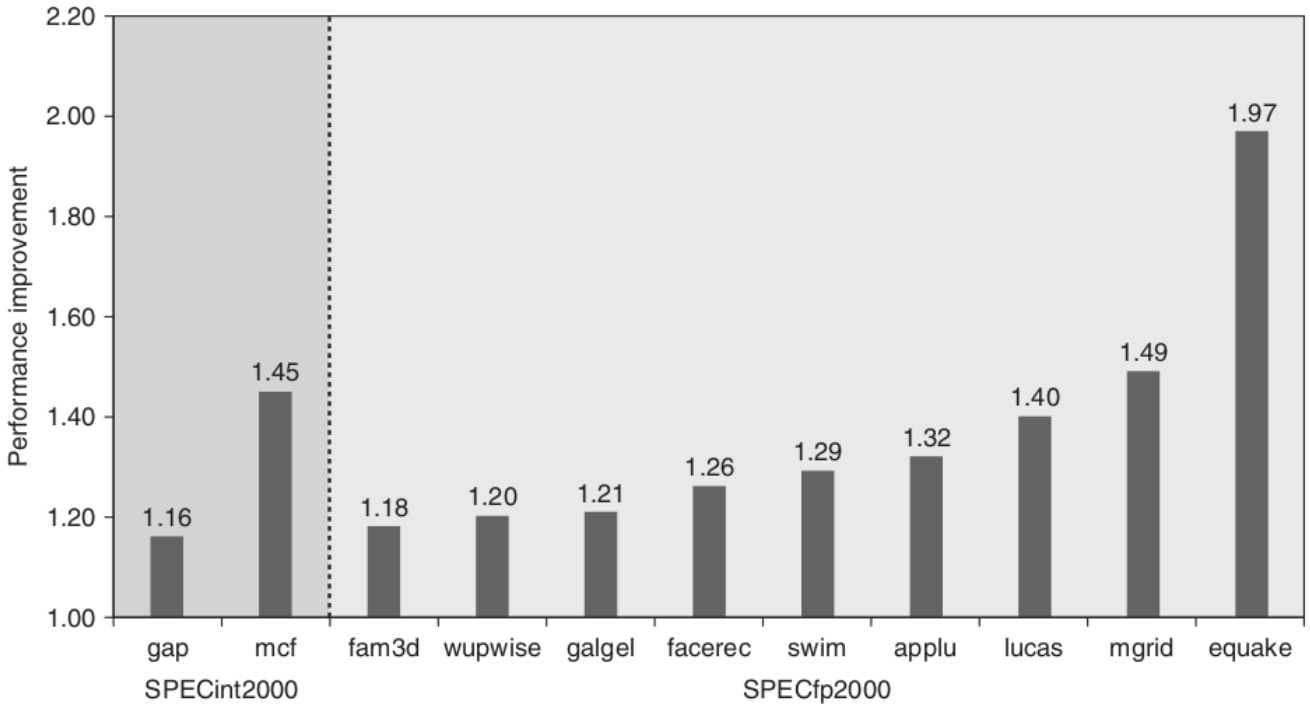
- **Critical word first** requests the missed word first from memory and send it to the processor as soon as it arrives; let the processor continue execution while filling the rest of the words in the block.
- **Early Restart** fetches the words in normal order, but as soon as the requested word of the block arrives send it to the processor and let the processor continue execution.

Merging Write Buffer

Write buffers are the basis of write-through caches. Furthermore, even write-back caches use a simple buffer for block replacement. In this idea, if the buffer contains other modified blocks, the addresses can be checked to see if the address of the new data matches the address of a valid write buffer entry. If so, the new data are combined with that entry.

Hardware Prefetching of Instructions and Data

This idea mainly emphasizes the ability to prefetch data and instructions form the main memory before they are requested by the processor. If prefetching is done correctly then the miss rate can be reduced significantly.



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

Compiler-Controlled Prefetching

An alternative to hardware prefetching is for the compiler to insert prefetch instructions to request data before the processor needs it. This can be done in two ways:

- *Register prefetch* will load the value into a register
- *Cache prefetch* loads data only into the cache and not the register.

This helps in reducing the miss rate in the same way as prefetching but with greater accuracy since the loaded blocks are hard coded by the compiler with certainty.

Conclusion

The techniques to improve hit time, bandwidth, miss penalty, and miss rate generally affect the other components of the average memory access equation as well as the complexity of the memory hierarchy. The existing techniques have been performing well but still the constantly evolving world Computer Architecture

needs even better optimization techniques to cope up with the ever increasing demand for improvedperformance.