Dana Toribio

Albert Cervantez

CPSC 440

8 December 2014

Applications of Software Optimization to Improve Cache Effectiveness

Abstract

Cache is the fastest form of memory in the memory hierarchy. Algorithm design deals with solving computational problems in the most efficient ways possible by working around the resource constraints. This paper brings together an overview of algorithm design and caching. Finally, applications of cache optimization using blocking algorithms are presented to provide examples of how algorithms and computer architecture are studied in real-world computing.

Overview of Algorithm Design

A very important topic in computation is algorithm design, a study that is not a stranger to computer architecture. Algorithms deal with solving computational problems in the most efficient ways possible. Fundamentally, “an algorithm is efficient when it consumes few resources”, wherein resources are defined as a “quantifiable [unit of measurement] that is expended when executing an algorithm” (Wortman 1).

Upon studying algorithm design at Cal State Fullerton, most of our study emphasized on *time* (measured as a unit of seconds, CPU instructions or generic steps) as our main computational resource. *Space* (measured in units of bits, bytes, gigabytes, or generic words) was also mentioned, but was not covered in depth as time. Aside from time and space, other resources that were mentioned included *input/output bandwidth* (measured in units of bytes or blocks), *cache misses* (measured in in units of integers) or *energy* (measured in units of kilowatt-hours). Computer Architecture called for more emphasis of space, input/output bandwidth and cache misses, which essentially brings our utilization of computational resources into full circle, minus the resource of *energy*.

Another important point that relates algorithm design to computer architecture is the idea that increased optimization gives way to more complex code. We will later see examples in the upcoming research samples that validate this point.

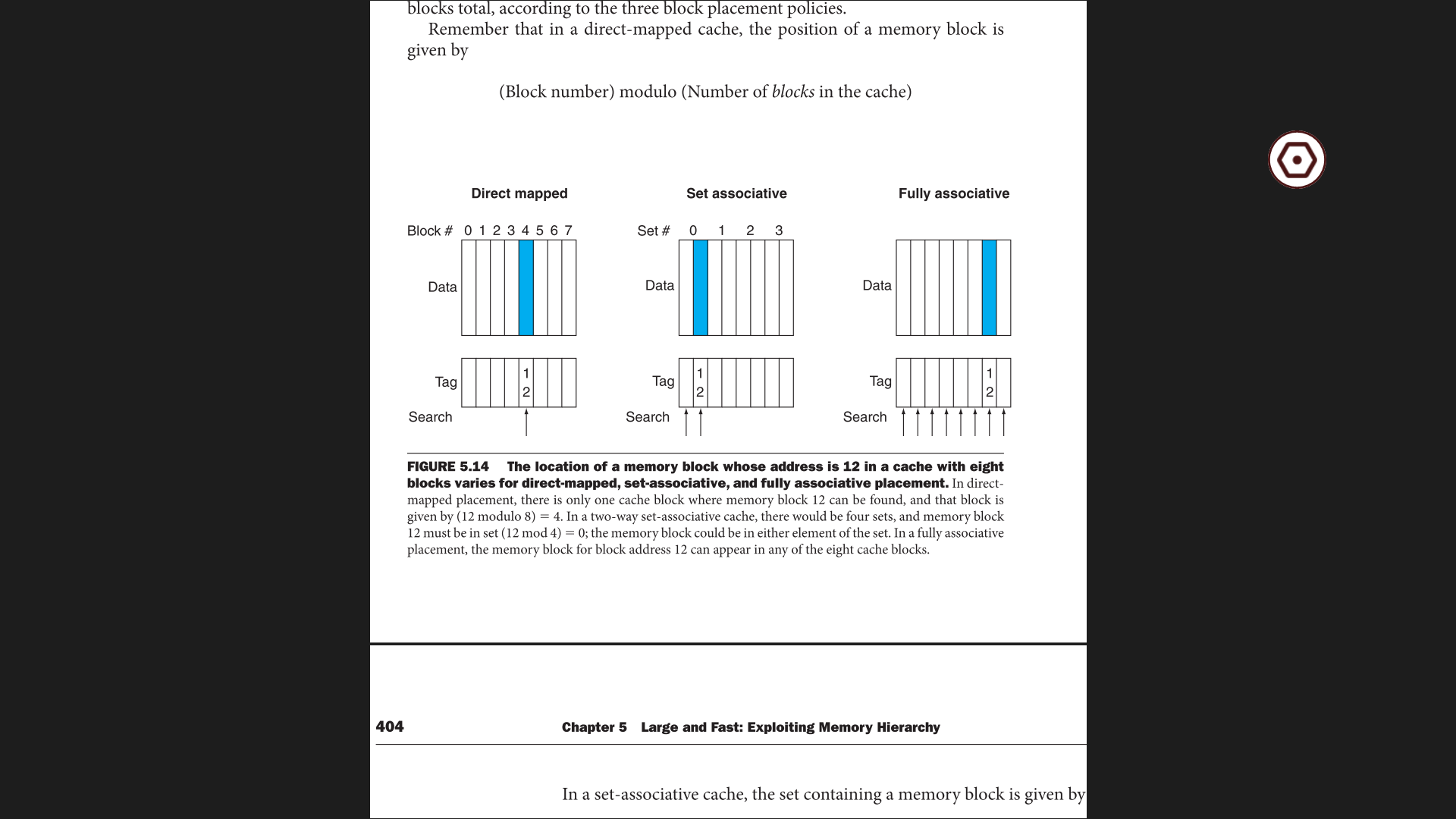
Overview of Caching

The main topic of Chapter 5 involves caching. A *cache* represents the “level of memory hierarchy between the processor and main memory” (Patterson 383). It also refers to “any storage managed to take advantage of locality of address” (Patterson 384). Essentially, addresses on cache are comprised of a *byte offset field* (lower 2*n* address bits, which associate with the location in the memory hierarchy table’s data field), *index* field (middle address bits, which associate with the index value in the memory hierarchy table) and *tag* field (upper bits, which associate with the tag field in the memory hierarchy table). Information in the address is matched to a table of memory hierarchy, which includes fields for the *valid bit*, *tag* and *data*.

Another major component of caching includes *cache hits* and *cache misses*. Cache hits occur when the tagged data is found within cache and is instantly accessed. Cache misses occur when the data is not already stored in cache, which in turn is populated into cache and made available for quick cache access. If all cache blocks are populated and a resulting cache miss occurs, the new cache data is stored in the least recently used (LRU) cache position.

Spatial locality is also a major point of interest in caching because it makes logical sense that if data is recently accessed, it is safe to assume that that specific data or data in really close proximity may be accessed again.

There are numerous cache structures known today, the most common being a *direct-mapped cache*, where each memory location is mapped to exactly one location in the cache. Other caching schemes include *fully associative cache* (a block can be placed in any location in cache) and *set-associative cache* (the cache has a fixed number of locations of at least two where blocks can be placed) (Patterson 384).



Trends in caching that occur show that miss rate increases the larger the block size is in relation to the cache size. This is caused by the number of blocks available in cache being limited due to the large block sizes being placed within the cache.

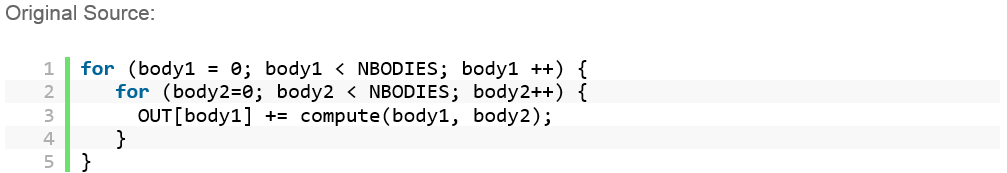
Overall, caching is an example of *prediction* since it relies heavily on spatial locality to find data in higher levels of the memory hierarchy.

Real-World Applications of Cache Blocking

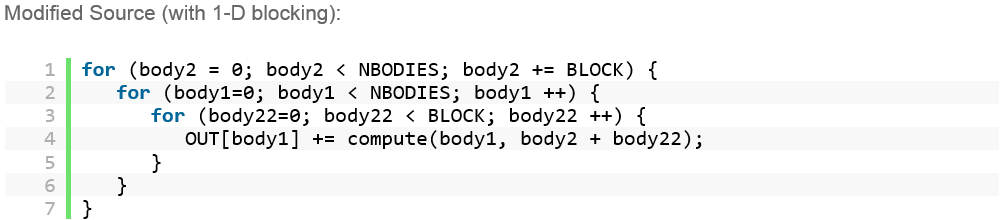
The practice of blocking algorithms is a well-known optimization technique. It is used for “improving the effectiveness of memory hierarchies” (Lam 63) as well as “blocking data structures to fit into cache” (CBT). Rather than operating on entire rows and columns of an array, blocking algorithms deals with sub-matrices known as a block, which in turn reuses data that are loaded into the faster levels of memory hierarchy. The blocking algorithm method relies on memory access data organization, which allows loads to cache with a small subset of a much larger data set. This small subset of data is then worked on. This use and reuse of data in cache reduces the number of memory accesses and in turn reduces pressure on memory bandwidth. This ties bandwidth as another resource relied on in the list of finite computational resources.

In relation to the algorithm design, it is commonly known that the simplest implementations are not always the most efficient. In instances of optimization, there tends to be an increased complexity in implementation as the major trade-off.

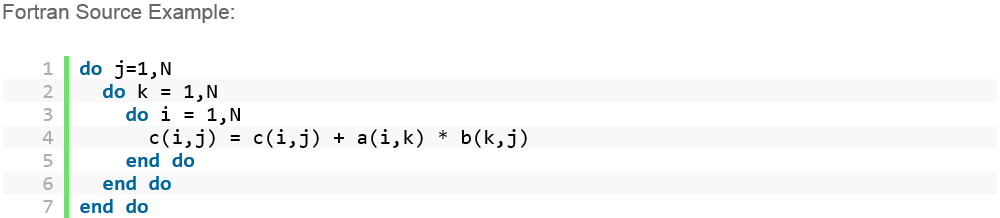
Examples given by the “Cache Blocking Techniques” article gives sound examples of the correlation between increased implementation complexities to increased optimization. Their first example includes optimizing a memory bandwidth-bound application with 1-dimensional blocking, which is initially implemented with two for loops but is bound to the memory bandwidth, thus allowing the application only to run at the speed of memory to CPU speeds, which was deemed “less than optimal” (CBT).



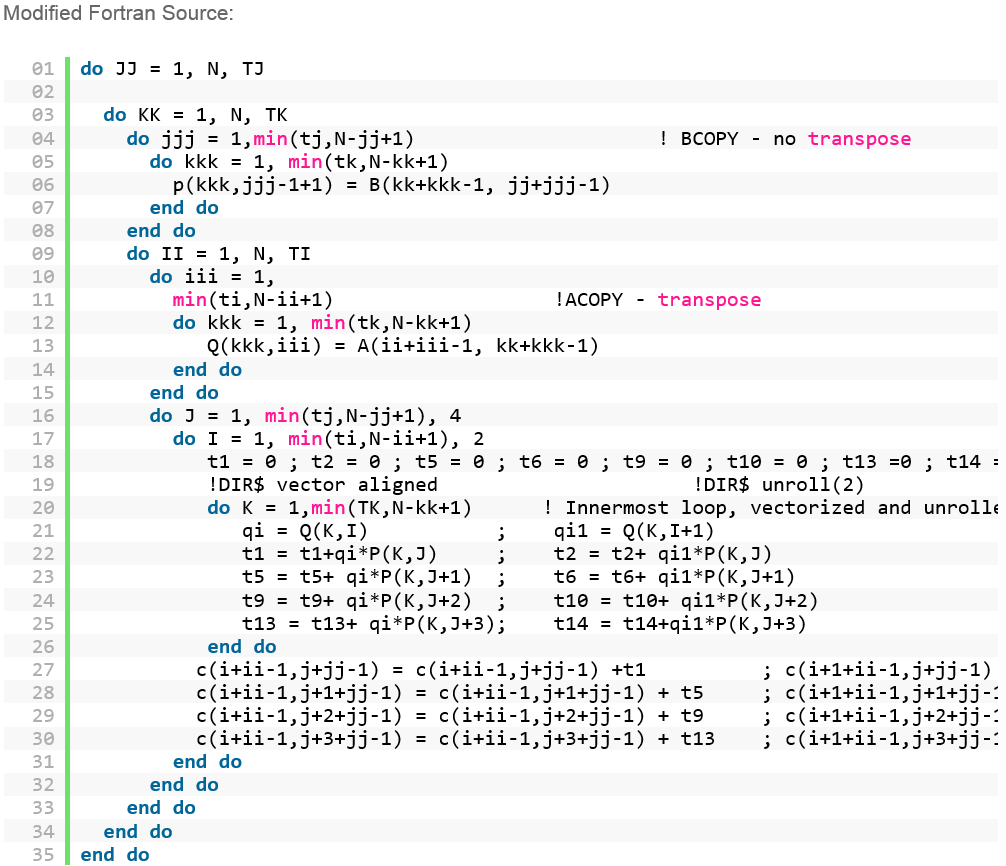
It was later optimized with three for loops, allowing a reuse in cache and better performance.



Further examples presented by the Intel article include a loop unroll-jam transformation implemented in Fortran with three loops with O(*n*3) efficiency.



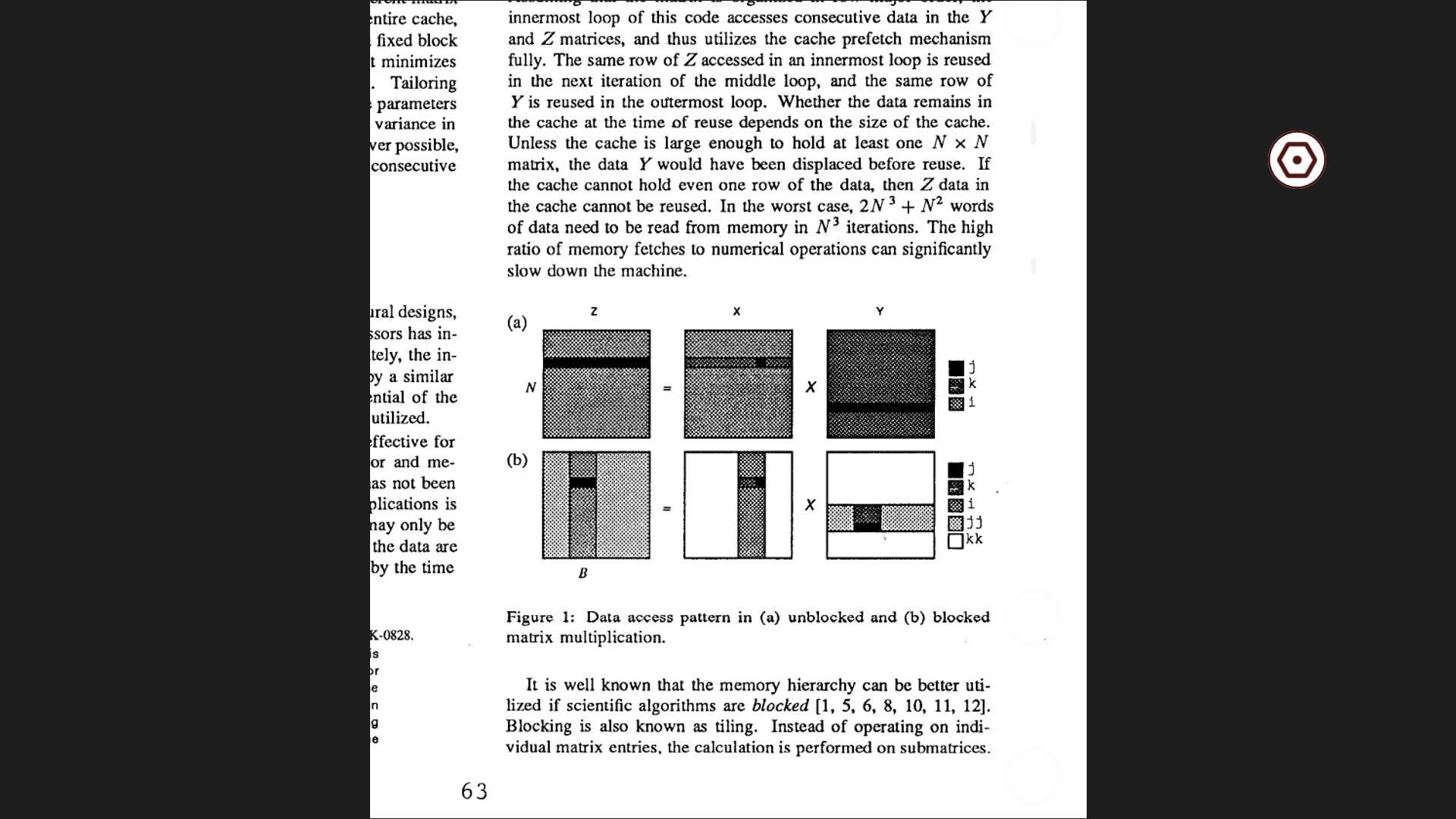
It is later optimized to allow advanced block-unroll-jam transformations which involve local copy-arrays.



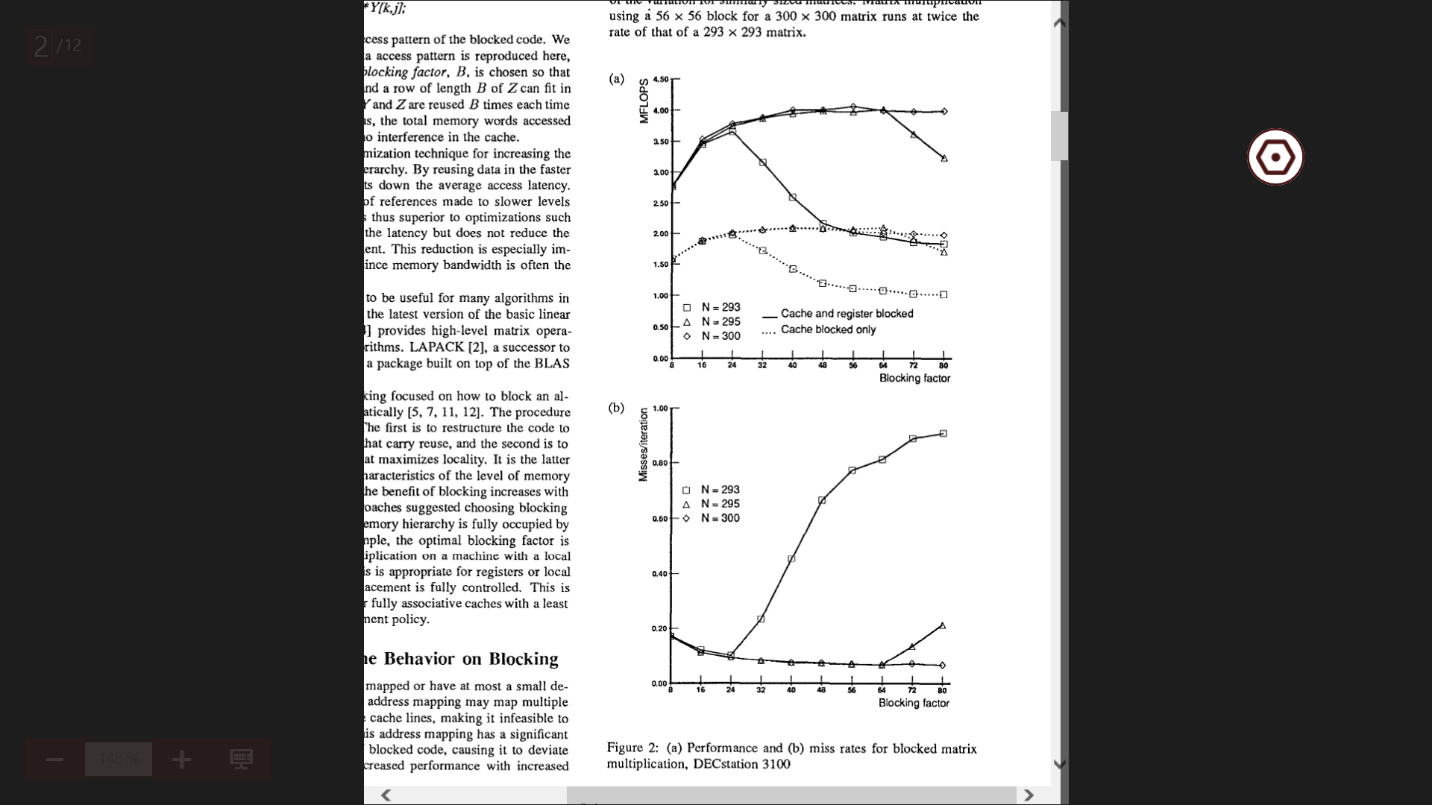
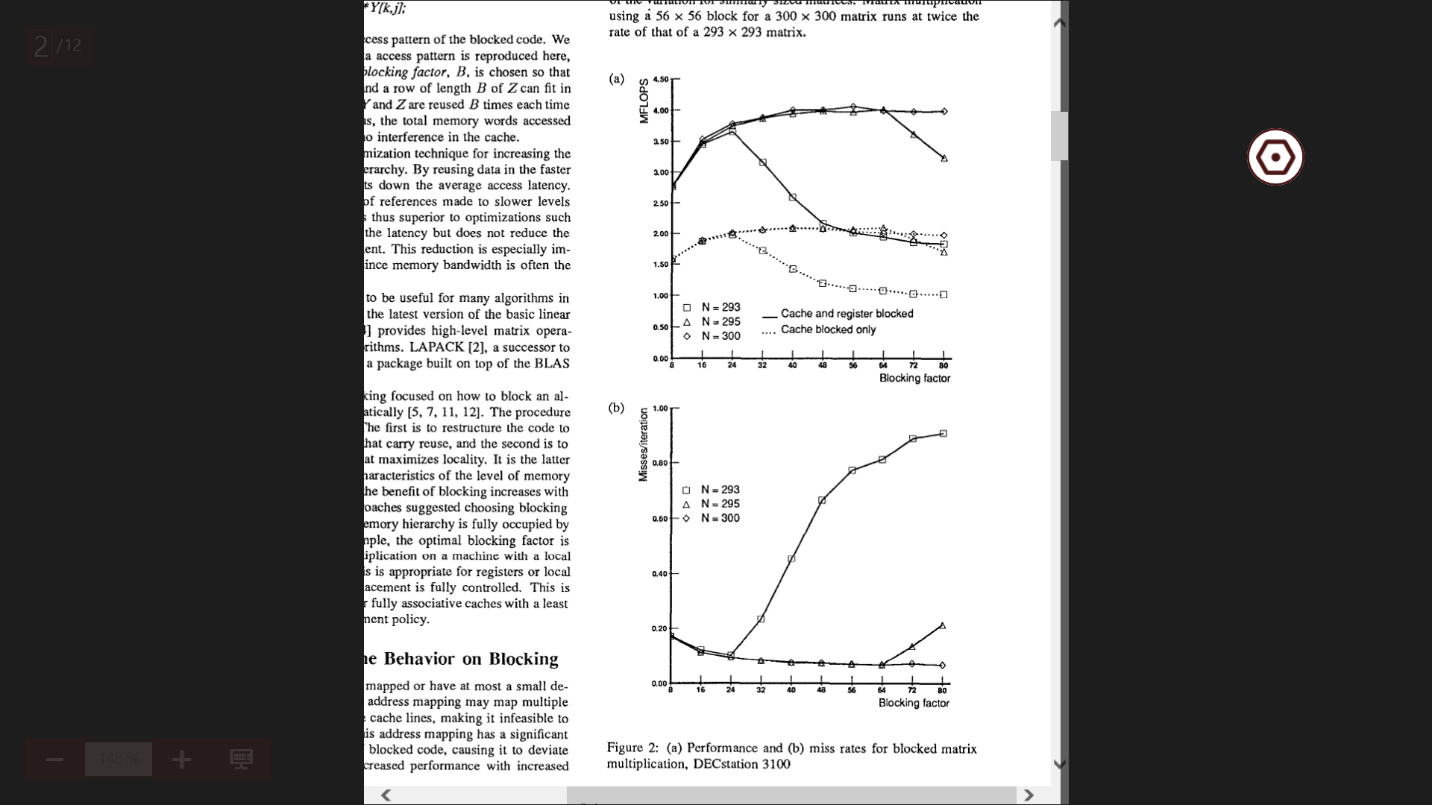
This results in an increased time efficiency of approximately O(*n*6), but is considered the “best performance” implementation (CBT), which is surprising considering the high time efficiency.

Other findings include block caching can be done in 1-dimensional, 2-dimensional or 3-dimensional spatial architectures. Temporal locality is relied heavily in cache blocking, which is known as temporal blocking. This helps to prevent bandwidth bottlenecks. This form of blocking allows the avoidance of repeated fetches of data stored within main memory.

In a paper by Lam, Rothberg and Wolf, their research presents cache performance data for blocked programs. Although the depth of their research is far outside the scope of this course, they provide many thorough empirical analyses of blocked algorithm experiments.



Through evaluating optimizations to improve performance, they make claims that blocking is superior to some optimization techniques like prefetching by showing several optimization of matrix multiplication in their report, similar to what was discussed in lecture.



Through a comprehensive analysis of the performance of blocked code on machines with caches, they were able to show that blocking is effective for reducing the memory access latency for caches. Also, applying blocking algorithms specifically to cache rather than to both cache and registers shows greater favor in lower MFLOPS for blocking algorithms applied only to cache.

Research provided by Nishtala, Vuduc, Demmel and Yelick gives an in-depth analysis of cache blocking in sparse matrix vector multiplication, an important computational kernel problem. In their research, they found that cache blocking is very important for some matrix and machine combinations, thus resulting in a speedup as high as 3x (Nishtala 1). They make aware the complications given by modern hardware architectures and the resulting overhead of data structure manipulation. Of the optimization techniques they covered were *register blocking*, *cache blocking* and multiplication with *multiple vectors*, with cache blocking being their main focus since cache blocking reorder memory accesses to increase temporal location and also adds an extra set of row pointers for each block (Nishtala 3). This study results in a trade-off that is needed to make between favoring adding temporal locality over the costs from increased overhead from data structures.

Conclusion

Extensive research and application in cache optimization through algorithms shows an affinity for both topics in computing. Understanding computer architecture has rounded out the points of interest in algorithm design that weren’t covered, especially in terms of resources. Through algorithm design and an understanding of computer architecture can further optimization of programs lead to greater performance and efficiency in program capability.

References

Intel Corp, Amanda S. "Cache Blocking Techniques." *Intel® Developer Zone*. N.p., 06 Dec. 2014. Web. 06 Dec. 2014. <https://software.intel.com/en-us/articles/cache-blocking-techniques>.

Lam, Monica S., Edward E. Rothberg, and Michael E. Wolf. "The Cache Performance and Optimizations of Blocked Algorithms." (n.d.): 63-74. Web. <http://suif.stanford.edu/papers/lam-asplos91.pdf

Nishtala, Rajesh, Richard W. Vuduc, James W. Dimmel, and Katherine A. Yelick. "Performance Modeling and Analysis of Cache Blocking in Sparse Matrix Vector Multiply." (n.d.): 1-70. Web. <http://bebop.cs.berkeley.edu/pubs/csd-04-1335.pdf>.

Patterson, David A., and John L. Hennessy. *Computer Organization and Design: The Hardware/software Interface*. Amsterdam: Elsevier Morgan Kaufmann, 2012. Print.

Wortman, Kevin, PhD. "Chapter 2: Efficiency Analysis." 2014. 1-29. Print.