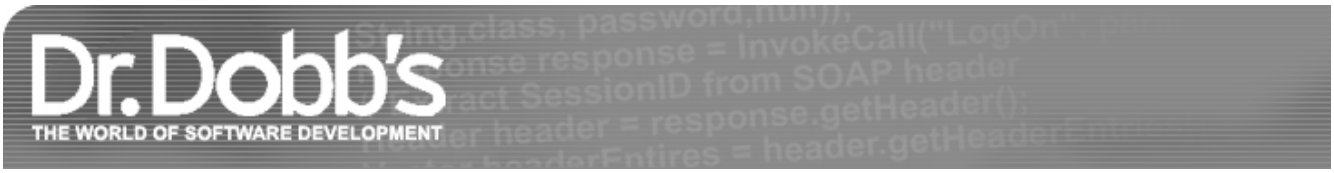

Cache-Friendly Code: Solving Manycore's Need for Faster Data Access

As the number of cores in multicore chips grows — Intel is poised to release the 50+ core Xeon Phi — ensuring that program data can be delivered fast enough to be consumed by so many processors is a huge challenge. Optimal use of processor caches is a key solution, and knowing these coding techniques will become a requirement.

November 12, 2012
URL:http://www.drdobbs.com/parallel/cache-friendly-code-solving-manycores-ne/240012736

The coming wave of manycore processors, such as the Xeon Phi, are many-headed beasts that can chew through a tremendous amount of data — but only if they are fed properly. If data can't be brought in quickly enough, some of the cores will starve and be forced to wait for data to arrive.

The multicore opportunity leads developers to focus on software scalability in unprecedented ways. Before, scalability was primarily of interest to those with access to unusual hardware. Now, it is a critical metric for assessing the long-term performance potential of any code or algorithm.

There are two ways to take advantage of multiple cores: Developers can run a single program and make it multithreaded, or they can run multiple processes at the same time. Scalability is an important characteristic in either case.

The case of a single program needing to be spread across multiple threads has been discussed numerous times in *Dr. Dobb's*, with extensive coverage of the various techniques.
The scaling of threaded programs is frequently limited by contention for resources. Resources that can limit scalability include processor cache space, memory space, and bandwidth limiting data transfer between different components (processor, memory, interconnect, and storage). With multi- and manycore processors, the key bandwidth limitation is often the link between main memory and the processor — the processor caches. These caches are critical for ensuring that the many cores of our beast are adequately fed.

Homogeneous Workloads

Other significant problems are composed of numerous distinct subproblems that are solved independently. In these cases, processor occupancy — the proportion of busy processors — can be raised by running many copies of the program at the same time. Each individual instance may not run terribly fast, but the workload as a whole should benefit from a net throughput that wouldn't be achievable on older processors. Examples of these workloads, often described as being "embarrassingly parallel," might be serving static Web content, calculating Monte Carlo financial models, and bulk processing of image or audio data.

However, when multicore processors are used to run many instances of simple programs, they are still subject to sharing system resources such as bandwidth, cache, and memory. These shared resources can still be bottlenecks even when the computations are completely distinct and embarrassingly parallel. When conflict occurs, cores must wait for data and so, they are idle part of the time. This can mean that even with embarrassingly parallel

workloads, adding tasks (even when cores are available to run them) will not necessarily result in faster throughput.

Heterogeneous Workloads

Virtualization is a way to achieve higher core occupancy at a system-administrative level because a large fraction of front- and back-office software is single threaded. Virtualization allows multi- and manycore processors to support wider workloads with a high degree of transparency to both the application and organization. But the scalability limits mentioned previously still apply, because when two or more virtual machines are placed on a multi- or manycore processor, all those processes share the same memory bandwidth and contend for the same cache space.

Cache and memory bandwidth are limiting factors in all these scenarios and they become progressively more challenging as the number of cores increases.

Cache Memory and a Serial Example

Allocation of space in the processor data cache is managed by the hardware and is based on the processor's access pattern. When the processor accesses a part of memory that is not already in the cache, it loads a chunk of the memory around the accessed address into the cache, in the hope that the processor will need that same data or adjacent data soon, and if it does, then that data will be in the cache and easily accessible.

"Cache" lines are the chunks of memory handled by the cache, and the size of these chunks is called the "cache line size." Common cache line sizes are 64 or 128 bytes.
The limited number of lines that a cache can hold is determined by the cache size. For example, a 64 KB cache with 64-byte lines has 1024 cache lines.

Prefetching in Hardware and Software

It is clear that when data is going to be used multiple times, keeping it in the cache will eliminate the latency problem for the second and subsequent accesses, but nothing discussed so far reduces latency the first time a program needs to access a given cache line. Ideally, the data would already be waiting in the cache for the processor to access the first time. This can be achieved through prefetching lines of data into the cache. Most processors have both hardware and software support for prefetching.

Software prefetching takes the form of instructions that don't operate directly on data, but tell the memory system to preload an address or range of addresses into the cache. Compilers can weave these instructions into computations as an optimization if it is likely that they will help.

Hardware prefetching watches where the program is reading from memory and, if it detects a pattern, it will start preloading what will be needed before the program asks for it. Hardware prefetchers work best with very regular access patterns.

Cache Coherency

Modern processor caches also maintain a property called "coherency." A coherent cache places a copy of data close to the core, then takes steps to ensure that the whole system functions as if there were only one copy of the data, therefore avoiding situations where separate cores see different values for what should be the same data. While vital for correctness, the mechanisms used to ensure coherency between items found in the caches of multiple cores can add substantial overhead for certain types of access patterns. Coherency mechanisms also need to be in place to avoid errors when direct memory access (DMA) technology is used to change data in memory without the CPU being involved.

Some Cache Examples

The amount of cache per processor core is generally limited, but much faster than main memory. For example, the quad-core Intel Sandy Bridge (Intel Core i7) has 32 KB of L1 (or innermost) cache per core, 256 KB L2

cache per core, and 2 MB L3 cache per core. Accessing the L1 memory takes 1.2 ns, L2 3.6 ns, and L3 requires 11.7 ns, respectively. AMD's 8-core Bulldozer (8150) has 16 KB/1024 KB/1024 KB of L1/L2/L3 memory per core, respectively, and has slower access times of 1.1 ns/8.4 ns/24.6 ns. Accessing main memory requires about 60-70 ns with either processor [1].

Consider the following simple routine:

```
static volatile int array[Size];
static void test_function(void)
{
    for (int i = 0; i < Iterations; i++)
        for (int x = 0; x < Size; x++)
          array[x]++;
}
```

This routine was run on a system with a cache size (L2) of 512 KB. For comparison purposes, the size of the array was varied while changing the number of iterations to keep the number of increments to the inner array constant. When the array size was greater than 512 KB, it took about 50 seconds to run. When the array size was smaller, so it fit completely within the cache, the calculation ran in about 5 seconds. The magnitude of this effect, approximately 10x speed up, is consistent with the ratio of latency to the cache and latency to main memory.

Multiple Processes, Cache Friendly Programs, and Scalability

So what can be done to fully utilize the performance of a multicore system when many separate processes are running on that system? Examining programs to see if they are using the cache efficiently is an important step. If cache-friendly algorithms are used, even in single-threaded applications, a foundation for efficient virtualization is established: A single greedy process can gobble up all the cache and leave other innocent processes to starve as they wait for every data element to be fetched from main memory. Let's look at the types of problems that can inadvertently make caches inefficient, then examine optimizations that can be made to simple programs to make them more cache friendly.

Inefficient Loop Nesting and Blocking

Dense, multidimensional arrays are common datasets in numerical applications. When a program is processing a multidimensional array, such as pixel data in image processing or elements in a finite-difference program, it is important to pay attention to which dimension is assigned to the inner loop and which dimension is assigned to the outer loop.

If an array is traversed along the wrong axis, only one element will be used from each cache line before the program continues to the next cache line. For example, consider this small snippet of C code iterating over a two-dimensional matrix:

```
double array[SIZE][SIZE];
for (int col = 0; col < SIZE; col++)
    for (int row = 0; row < SIZE; row++)
        array[row][col] = f(row, col);
```

In one iteration of the outer loop, the inner loop will access one element in each row of a specific column. For each element touched by the loop, a new cache line is accessed. If the array is larger than the cache size, the rows from the top of the matrix may already have been evicted from the cache when the bottom of the matrix is reached. The next iteration of the outer loop then has to reload all of the cache lines again, each time paying the full main-memory latency cost.

If the nesting of the loops is changed so that they traverse the matrix along the rows instead of along the columns, the program receives a lot more benefit from the cache:

```
double array[SIZE][SIZE];
    for (int row = 0; row < SIZE; row++)
         for (int col = 0; col < SIZE; col++)
             array[row][col] = f(row, col);
```

As the program processes each column along a row, data is moved into the cache and all of the data in that line is used before the cache line ends up being discarded. Accessing data sequentially also allows hardware prefetch logic to engage, placing data in the cache before the processor even asks for it. This maximizes the use of available memory bandwidth. A code transformation like this can result in performance improvements of up to 15x for large array sizes. For example, the LBM code of the SPEC2006 benchmark suite contains such a performance problem and was made to run 2.7-times faster on a quad-core when the simple code transformation suggested above was applied to one of its loops.

Sometimes it isn't possible to choose a loop order that puts all the data that needs to be read and written adjacent to one another. If this is the case, then a more-advanced technique called "blocking" can be used. Blocking involves breaking up the computation into a set of subcomputations that each fit within the cache. Blocking does involve reordering the sequence of computations and it is important to verify that this reordering doesn't violate a data dependency.

Padding: The Bane of Cache Line Utilization

Many modern processors require memory allocations to be made to aligned addresses. This generally means that a field must be stored at an address that is a multiple of its size; for example, a 4-byte field must be stored at an address that is a multiple of 4, and an 8-byte field must be stored at an address that is a multiple of 8.

Compilers insert padding between variables and fields to ensure that each field starts at an address with correct alignment for its particular data type. This padding consumes valuable cache space and memory bandwidth. Consider the following code snippet:

```
struct record {
    char a;
    int b;
    char c;
};
```

Assume that `char` is a 1-byte data type and `int` is a 4-byte data type. The compiler will then lay out data with three bytes of padding between fields `a` and `b` to ensure that `b` is stored at an offset that is a multiple of 4. The overall data structure will take up 9 bytes.

If a developer moves the field `a` after `b`, the alignment requirements of all fields can be satisfied without any padding. That reduces the overall size of the structure to 6 bytes. The simplest way to ensure the compiler does not add padding is to sort the fields by their alignment requirements. Start with the fields with the greatest alignment requirements and then continue in declining alignment requirement order. Clearly, this dense way of allocating the data structure results in both a leaner usage of the cache capacity and a lower bandwidth demand because less padding data needs to be transferred from the DRAM. These kinds of simple transformations, such as reordering fields to increase data density, can result in significant performance improvements.

Moving Unnecessary Data

Another type of poor cache line utilization can be seen in the following code snippet containing a vector of data objects, each being a structure with a status and a payload:

```
struct object_def {
    char status;
    double object_var1, object_var2, object_var3, object_var4, object_var5 ;
};

struct object_def object_vector[Size];

void functionA()
{
    int i;
    for (i=0; i < Size; i++) {
        if (object_vector[i].status)
           do_something_with_object(object_vector[i]);
    }
}
```

The program periodically goes through all the elements of the vector and checks each element's status. If the status indicates that the object is active, some action is taken and the object variable is accessed. The status byte and the five object variables are allocated together in a 48-byte chunk. Each time the status of an object is checked, all 48 bytes are brought into the cache, even though only one byte is needed. A more efficient way to handle this would be to have all the status bytes in a separate vector, as shown below, which reduces the cache footprint and bandwidth needed for the status scanning to one-eighth of the previouus code example:

```
struct object_def {
     double object_var1, object_var2, object_var3, object_var4, object_var5 ;
};

struct object_def object_vector[Size];
char object_vector_status[Size];

void functionA()
{
    int i;
    for ( i=0; i<Size; i++) {
        if (object_vector_status[i])
           do_something_with_object(status[i],object_vector[i]);
    }
}
```

Performance improvements of a factor of two have been achieved for the SPEC benchmark suite Libquantum, and a factor of seven for the open-source application cigar when similar optimizations were applied.

Other Cache Issues

Other program behaviors that can limit cache effectiveness (and may represent an occasion for optimization) include:

- Using random access patterns, such as linked lists
- Using unnecessarily large and sparse hash tables
- Fetching or writing data into the cache even though it is unlikely to be reused
- Updating only a single byte of a cache line
- Prefetching data that will never be used
- Prefetching data into the cache too early or too late
- Separating computations on large arrays into different loops when there isn't a data dependency that requires one loop to be completed before the other one starts

Programmers will need to consider the implication of these behaviors in order to create applications that run fast and efficiently, and have the scalability needed to take advantage of multi- and manycore processors. While compilers can recognize some of these patterns and apply some optimizations automatically, there are practical limits to what can be detected at compile time.

Caching with Multithreaded Programs

Multithreaded programs can allow a single computation to utilize more than one core in a multicore program. When developers program with threads, they create multiple execution contexts within a single operating system process. Each execution context can run simultaneously on a separate core. The process itself has a memory image and the threads operate on that memory image simultaneously. Communication between the threads can happen very naturally by having them read from, and write to, shared data structures.

The cache is involved in almost all movement of data between the cores of a multicore processor and always works with data in units consisting of cache lines. Programming with threads is complex enough that it is easy to neglect the cache and operate under the assumption that the data that is moved back and forth between processors at a finer granularity of individual words and bytes. This failure can easily result in a bottleneck that limits scalability.

One way that this conceptual challenge manifests is false sharing, in which two cores unwittingly write data items to the same cache line — thereby creating excess coherency traffic as the cores update their respective

copies.

Lessons from Cache Optimization and General Parallelism

The observations that guide cache optimization are closely related to some of the key concepts for thinking about parallelism more generally. Cache optimization draws attention to two aspects about the program data.

The first aspect is the way data is used. Is it used repeatedly within sequential iterations of a calculation or used only one time? How is the program looping over this data? Is there a way of walking through the data that keeps computations performed on the same data close together? Is a given location needed by multiple threads? This awareness of the relationship between data and computations is essential to finding opportunities for parallelism.

The second aspect is the way the data is placed on cache lines. Object-oriented programming encourages programmers to abstract away details such as data placement. This abstraction allows for reuse, but obscures important details that are critical to understand and optimize performance. Awareness of bandwidth and data placement is fundamental to cache optimization and is key to scalability in parallel programming.

Conclusion

Running many programs at the same time or single programs with many threads is necessary to maximize the benefit of multi- and manycore processors. This means that the power of next-generation, manycore beasts can be utilized only if the cores can be supplied with a steady diet of data.

The processor cache plays a vital role in feeding the multicore beast. Ideally, it places the required data where it can be easily and quickly fetched, keeping the many cores of the processor fed. When the cache isn't able to function efficiently, the data is not quickly available, and application performance can slow to a crawl. This article has looked at how cache works and discussed a variety of different techniques that can help developers create programs that make better use of the cache. As cache optimization can be both daunting and complex, developers should take advantage of cache memory optimization tools that analyze memory bandwidth and latency, data locality, and thread communications/interaction in order to pinpoint performance issues and obtain guidance on how to solve them.

At the algorithm design level, there are two different approaches that can be taken towards designing cache-friendly programs. Cache-aware algorithms factor in details about the cache as design inputs and are aggressively tuned to perform well at a specific cache size. Cache-oblivious algorithms are tuned for caches more generally and not for a specific cache size. That both of these techniques can yield highly efficient algorithms is not a surprise, since we've seen here what kind of a difference caches can make.

Programmers need these techniques and tools if they want to be able to use their software efficiently on future generations of processors. Those who neglect the cache are likely to end up with programs that try to monopolize it and end up starving other programs sharing that cache.

References

[1] Measurements come from benchmarking stories on the AnandTech and ExtremeTech websites.

Chris Gottbrath is the Principal Product Manager for TotalView and ThreadSpotter at Rogue Wave Software.