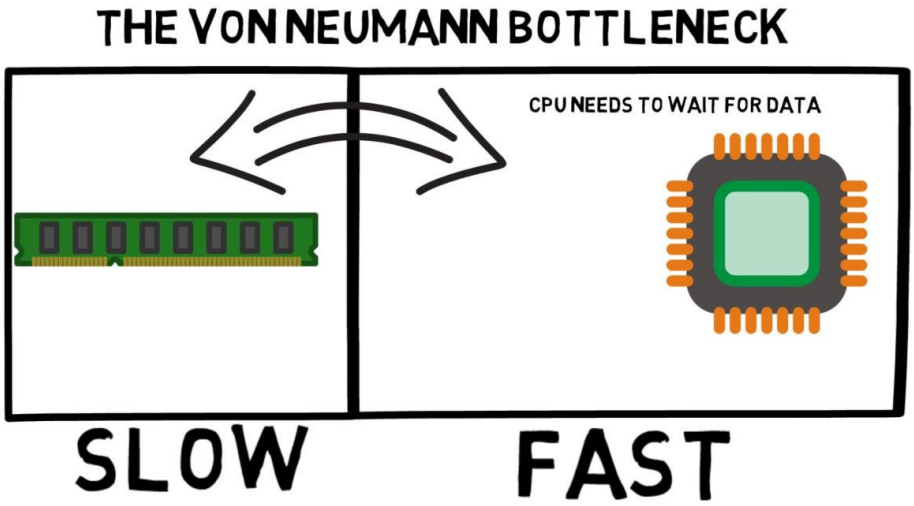


Make your programs run faster by better using the data cache

May 22, 2020 / Performance / Leave a Reply

Developers are confronted all the times with the need to speed up their programs and the most obvious way is to come up with a fancy new algorithm with a lower complexity. Instead of $O(n^2)$ complexity our new algorithm has a lower complexity of O(n log n) and we can carry on happily to our next challenge. This is the best way to go, but often not the possible way. What now? Is there a way to squeeze out more performance out of our existing algorithm. Well actually there is. It's called: low-level optimizations.

Like what you're reading? Follow us!

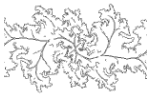
Recent posts

Flexibility and Performance

Speedscope: visualize what your program is doing and where it is spending time

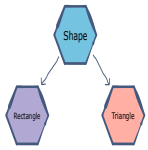
Speeding up an Image Processing

First a little bit about low-level optimizations. Low level optimizations are all about how to best exploit the particularities of the underlying architecture to get better performance. This is the first post in the line of posts that will deal with low-level optimizations. We will explore some ideas on how to better leverage memory cache subsystems. For those who are already familiar with memory cache in modern day multiprocessor systems, feel free to skip *Data Cache* chapter.



Algorithm

The true price of virtual functions in C++



2-minute read: Class Size, Member Layout and Speed

About me

Johny's Software Lab LLC is a software development consulting house specializing in software performance. If you or your developer team needs help on how to make your C or C++ program run faster, or you are missing out important deadlines because of problems with

Table Of Contents

Data Cache

Computer systems in general consist of processor and memory. In modern day systems memory is hundreds of times slower

performance, this is the right place to be.

than the processor, so processor often has to wait for memory to deliver the data. Clever hardware engineers have come up with a solution to offset the difference in speed: they add a small yet very fast memory called cache memory[1] that compensates for the difference in speed. When the processor wants to access data in the main memory, it first check if the data is already present in the cache memory, and if so, it gets its data very fast. Otherwise it will have to wait for the main memory to provide the data and this involves a lot of wasted processor cycles.

Normally cache memory is split into instruction cache memory and data cache memory. The purpose of the first is to speed up access to instructions and the purpose of the second is to speed up access to data used by instructions. In this article we are concerned only about how to speed up your program by better using the data cache memory.

Why does the memory cache make the system run faster?

So why does adding cache memory works? After all, the program can access any memory location at any time, therefore the data should never even be in the cache. In theory yes, but in practice accessing memory locations in random fashion is something real programs almost never do.

There are two principles that govern the behavior of the real world programs. The first is called *temporal locality* and it essentially means that if the processor is currently accessing a certain memory address, there is a high probability it will access the same memory address in the near future (think a counter in a loop). The second is called *spatial locality* and what it means is that if the processor is currently accessing a certain memory address, there is a high probability it will access neighboring memory addresses in the near future (think running through arrays).

Data Cache Internal Organization

Let's now look at how cache memory looks from within. Cache memory is divided into cache lines and in modern processors each cache line can typically hold 64 bytes of data. One cache line corresponds to one 64 byte block in the main memory. Access to one byte within a 64 byte memory block means that the whole 64 byte memory block will be loaded into the cache line. When the block is fetched into the cache line, a mapping is created between the cache line and the original memory block. All accesses made by the processor to the same memory block will be served from the cache. When the cache line is not used for a long time or cache needs to make place for new data, the cache memory returns the modified block back to the main memory. Notice that this happens like magic and programs are not aware of it.

Tips for making your program run faster

So, now when you have an overview of how data caches work, let's move on to some actual tips on how to better exploit your data cache in order to make your programs run faster.

A note: in the following chapters, I will use the term *array* for both traditional C style arrays and C++ `std::vector` and `std::array` classes. Also, I will use the term *class* for both C style `struct` and C++ style `class`.

Tip: When accessing data linearly, use vectors or arrays

Linked lists, hash maps, dictionaries etc. are great data structures for many things, but they are not cache friendly. Iterating through such a data structure involves many cache misses. If performance is important, stick to arrays. If that is not possible, try to use more exotic data structures that combine cache efficiency of the arrays and flexibility of other data structures. Gap buffer is an example of one such data structure. It is a combination of arrays and linked lists, and it allows excellent cache efficiency combined with ability to cheaply insert

or remove elements. Another one is Judy array, a tree implementation of a sparse array that is cheap to insert and remove elements and which is cache-friendly.

Tip: Variables you access often together should be close to one-another in memory

If there are several variables that are accessed together, they should be declared one after another. This increases the likelihood that the other variable will already be in the cache after the processor has accessed the first variable, thus avoiding cache misses.

Consider following class:

```
1.  class free_memory_list {
2.    void* head;                 /// Pointer to the beginning of
    the list
3.    Statistics statistics;      /// Statistics about list usage
4.    int count;                  /// Number of elements in the list
5.    Allocator* base_allocator;  /// Pointer to the class used for
    memory
6.                                /// allocation and deallocation
7.  };
```

This class implements a linked pointer list. If our program uses that class in such manner that it access variables `head` and `count` as a bundle, then they should be placed one after another in the class definition. In that case we increase the probability they will actually be in the same cache line.

Tip: Use array of values instead of array of pointers

First idea when comes to mind when speaking about arrays of classes or structs is to use pointers instead of values. This solution has many advantages over arrays of values, including run-time polymorphism and less memory usage in case of unallocated elements in the array, but with a performance penalty. Accessing the variable using a pointer invariably involves a cache miss. So for fast array access dispense with the pointers and go with values.

So now when we have array of classes as values, we have things going for us. Every time we access an element in the array, the cache prefetcher will get more elements that are close to the one we are currently accessing. If we are accessing elements of the array that are adjacent to one another, the data cache is maximally utilized.

Tip: Optimize access to array of classes or structs

If we are accessing elements of the array in random fashion, we can expect some cache misses. But we can have more or less misses depending on how we organized data in our class. Example:

Let's assume we have a class `my_class` and let `sizeof(my_class)` equals 48. First element of the array starts at offset zero, second element of the array starts at offset 48, third element start at offset 96 and fourth element at offset 144. If our cache has cache lines size of 64 bytes, this means the first element will fit cache line zero (bytes 0-47), second element will be split between cache line zero and cache line one (bytes 48 – 63 go to cache line zero and bytes 64-95 to cache line one), third element will be split between cache line one and cache line two (bytes 96-127 go to cache line one and bytes 128-143 go to cache line two) and fourth element will fit the third cache line (144 – 191).

In case there is a random access to the elements of the class, having one element split between two cache lines can be bad from the perspective of data cache utilization. The cache memory will need two accesses to the main memory in order to read a single element. So how to avoid it? How to make each element fit the minimal number of cache lines? Here are the rules:

- Size of class needs to be a multiple of cache line size
- Starting address of the array needs to be a multiple of cache line size

To make the size of class a multiple of cache line size, we can either manually add padding or ask the compiler to do that for us, C++11 allows this with `alignas(64)` specifier. If this is not available GCC/CLANG compiler offers `__attribute__((aligned (64)))`.

To make the starting address of array a multiple of cache line size, we can either allocate a bit more memory than we need and then manually determine the start of the array so that the array is correctly aligned. Better solution is to ask the compiler and libraries to help us; we could use `posix_memalign` to allocate aligned memory on the heap, or `alignas(64)` and `__attribute__((aligned (64)))` for aligned memory on stack and global memory. Here is an example on how to manually allocate array of classes using `posix_memalign`:

```
1.  my_class* array_of_my_class;
2.  posix_memalign((void**)array_of_my_class, 64, SIZE *
    sizeof(my_class));
3.  for (size_t i = 0; i < SIZE; i++) {
4.      ::new (&array_of_my_class[i]) my_class(i);
5.  }
```

The syntax looks a bit scary but it actually isn't. We declare a pointer to `my_class` that we will use to hold the allocated array. Next we allocate memory for the array using `posix_memalign`. We specify the alignment parameter to 64. And finally, in loop, we call the constructor for each element of the array. Notice that we are using `::new` operator, this operator doesn't do the memory allocation, instead it executes the constructor on the piece of memory provided as an argument.

Tip: Access data in your matrices efficiently

If your program works with matrices, you need to be aware how matrices are stored in memory. Matrices are by definition two dimensional, whereas memory is one dimensional. C and C++ compilers lay out matrices row by row. What this means is if we access an element of the matrix, several following elements in the same row will be available in the data cache as well.

This seems trivial, but it can have a profound effect on performance. Consider the simple matrix multiplication algorithm:

```
 void multiply_matrices(int in_matrix1[][N], int in_matrix[][N],
int result[][N])
 {
     int i, j, k;
     for (i = 0; i < N; i++) {
         for (j = 0; j < N; j++) {
             result[i][j] = 0;
             for (k = 0; k < N; k++) {
                 result[i][j] += in_matrix1[i][k] *
                                 in_matrix[k][j];

         }
     }
 }
```

Courtesy of https://www.geeksforgeeks.org/c-program-multiply-two-matrices/

It runs through `in_matrix1` row-wise and through `in_matrix2` column-wise, Now, `in_matrix1` works fine with regards to caching, but`in_matrix2` is a disaster for the cache. Every access to next element in `in_matrix2` results in cache miss, and even though not easily visible, there is a large performance penalty to this simple solution.

And the fix? It's super simple. Transpose `in_matrix2` before sending it to multiplication, and then go through both matrices row-wise.

Tip: Avoid padding in your classes and structs

A small note about data alignment: for all primitive data types, if data type is 4 bytes in size, its starting address needs to be divisible by 4. If data type is 8 bytes in size, its starting address needs to be divisible by 8. If the variable of size N starts at address that is divisible by N, we say that the variable is *correctly aligned* or just *aligned*, otherwise it is *unaligned*. An alignment requirement is often made by hardware and it is enforced by the compiler as much as possible. Now moving on.

To make sure that data in your classes and structs are correctly aligned, C and C++ compilers can add padding: these are unused bytes added between members of your class to make sure all the members are correctly aligned. Consider following example:

```
1.  class my_class {
2.    int my_int;
3.    double my_double;
4.    int my_second_int;
5.  };
```

One would expect that the size of this structure is `sizeof(int) + sizeof(double) + sizeof(int) = 16`. However, double needs to be eight bytes aligned, so after the member `my_int` compiler adds four bytes of padding so that `my_double` is correctly aligned. So we arrive to 20 bytes.

Additionally, in order to make the class data correctly aligned in arrays, the class takes the alignment requirements of the member with the highest alignment requirement. And the size of class needs to be a multiple of its alignment. In above example, ints have alignment requirements of 4 and doubles have an alignment requirement of 8, therefore our class needs to be 8 bytes aligned. And since the size of class needs to be multiple of its alignment, the compiler adds additional 4 bytes of padding at the end of the class, so the size of the class goes up from original 20 to 24 bytes.

How does the padding influence the cache efficiency? Let's say our cache memory has a 64 byte cache line. In our example, only 2.7 instances of `my_class` can fit a cache line, as opposed to four instances that we would expect without padding. Also, padding bytes are loaded into the cache memory, but your program never uses them.

So, in order to better use the cache, sort the variables in the declaration of your classes by size from largest to smallest[2]. This guarantees that compiler will not insert any padding and that the program will better use the data cache. Here is the same

class with slightly modified order of members that avoid padding:

```
class my_class {
  double my_double;
  int my_int;
  int my_second_int;
};
```

Size of this class is now 16 bytes and four instances fit a single cache line.

There is a tool you can use to explore the paddings in your classes called *pahole*. It needs to be built from sources since the version in the repositories doesn't support C++11.

Tip: Use smaller types if applicable

One of the ways to avoid padding in classes and therefore fit more data in the data cache is to use smaller types. Sometimes we declare four integers, but actually four short integers would suffice. Or maybe you have several bool variables in definition of your classes that you use to hold various flags. Instead of using bool, you could use chars or bool bit field, e.g:

```
class my_class {
public:
    bool my_bool1:1;
    bool my_bool2:1;
    bool my_bool3:1;
    bool my_bool4:1;
    int my_int:1;
};
```

In the above example, each bool takes one bit. But the compiler will insert padding after `my_bool4` in order to correctly align `my_int`. This can be avoided by rearranging the order of members of `my_class`.

Another example: one 64 byte cache line fits eight integers or four long integers. If you have an array of 1M elements, integer version takes 4MB whereas long integer version takes 8MB. It is obvious that loading 8MB into the cache is slower than loading 4MB.

But beware! Processors internally work best with native word sizes (e.g. 8 bytes in modern 64 bit architectures, or less in architectures in embedded systems). Using smaller data types might even decrease the speed, so you will need to measure to get the exact performance difference

Tip: Avoid heap allocation if stack could do

Heap is inefficient for several reasons:

- Calls to `malloc` and `free` are slow
- Access to those locations is indirect and it will result in cache misses more often

On the other hand, top of the stack is almost always in cache and super fast to allocate and deallocate. There are several tricks you could use to speed things up. If your program needs to allocate variable sized array, consider allocating array on the stack instead of heap (GCC but also some other compilers support this). If your program needs to allocate a lot of small memory blocks using `malloc`, consider allocating one large block and then splitting it into smaller according to your needs. If your program allocates and deallocates many objects of the same type, consider caching memory blocks instead returning them with `free`, so they will be quickly to allocate if later needed.

Tip: Use your data while still in cache

Ideally we would like to load data from the memory to the cache exactly once, do some modification on it, and then return them back to the operating memory. If you need to fetch the same data two times, you are not using the cache optimally.

Example of finding a minimal and maximal element of the array.

```
1.  int * a = initialize_array(size);
2.  int min = find_min(a, size);
3.  int max = find_max(a, size);
```

We have calls to two functions here, one that finds maximum and one that finds minimum in the array. Each function has its own loop, and inside the loop it iterates through the elements of the array. Assuming array is big enough, the elements of the array will be loaded into the cache two times.

Solution is simple: do all the work on the array inside one loop. Here is the corrected version:

```
int * a = initialize_array(size);
int min = a[0];
int max = a[0];
for (int i = 0; i < size; i++) {
  min = std::min(a[i], min);
  max = std::max(a[i], max);
}
```

Here we iterate through the array only once. Array data are loaded to the data cache only once, which utilizes the data cache much better.

Tip: Avoid writing to memory if possible

All writes to memory go through the data cache[3]. When a write is made, the cache marks that cache line as “dirty”. If a cache line is dirty, that means that it is different from the content of the memory and sooner or later its content will have to me written back to memory. This causes the slowdown. Check out these two algorithms:

```
void sort_fast(int* a, int len) {
    for (int i = 0; i < len; i++) {
        int min = a[i];
        int min_index = i;
        for (int j = i+1; j < len; j++) {
            if (a[j] < min) {
                min = a[j];
                min_index = j;
            }
        }
        std::swap(a[i], a[min_index]);
    }
}

void sort_slow(int* a, int len) {
    for (int i = 0; i < len; i++) {
        for (int j = i+1; j < len; j++) {
            if (a[j] < a[i]) {
                std::swap(a[j], a[i]);
            }
        }
```

```
        }
    }
```

The above code shows two similar functions for sorting numbers. They both work the same way. They find the smallest element and put it at the position zero, then they find the next smallest element and put it at the position one etc.

Function `sort_slow` looks for the element of the array that is smaller than the `a[i]` and if found immediately swaps them. It continues to swap elements every time an element is found that is smaller then `a[i]`. Function `sort_fast` looks for the element of the array that is smaller than `a[i]` but it doesn't do swapping, instead it keeps the new smallest element in a temporary variables `min` and `min_index` (compiler probably uses a register for these temporary variables). When the function finishes running through the array and has found the ultimate smallest element, only then it replaces the content of `a[i]` and `a[min_index]`. Function `sort_fast` is two times faster than `sort_slow` on my system.

Tip: Align your data properly

Your variables need to be aligned properly. This makes sure that the whole variable is located in a single cache line, as opposed to being split between two cache lines. If your system doesn't support access to misaligned variables, things could get very slow because your operating system might emulate the access to misaligned memory addresses.

Most of the time compiler makes sure the data is aligned properly, but easygoing developers might create places with misaligned memory accesses. This often happens when converting pointers from one type to another. Example of a bad alignment:

```
unsigned char serialized_data[1024];
read_data(serialized_data);
int* header_pointer = (int*) (serialized_data + 3);
int header = *header_pointer;
```

In this example, we convert char pointer to int pointer thus making `header_pointer` misaligned. Dereferencing `header_pointer` creates a misaligned memory access. On some architecture the program would crash, on others it will slow down. A corrected example:

```
unsigned char serialized_data[1024];
read_data(serialized_data);
int* header_pointer = (int*) (serialized_data + 3);
int header;
memcpy(&header, header_pointer, sizeof(int));
```

Here, we use `memcpy` to copy the value from the input array to header variable. Function `memcpy` doesn't need a proper alignment and this code better exploits the data cache and it is portable.

Tip: Use software prefetching

If your algorithm does not access its data one by one, but instead jumps around the memory in random fashion, you can use software prefetching to tell the processor which data you will be accessing so it has time to load them into cache before they are needed. For example, GCC and CLANG compilers offer *__builtin_prefetch* builtin that allows software prefetching. Here we give an example of binary search algorithm:

```
int binarySearch(int *array, int number_of_elements, int key) {
    int low = 0, high = number_of_elements-1, mid;
    while(low <= high) {
        mid = (low + high)/2;
#ifdef DO_PREFETCH
        // low path
        __builtin_prefetch (&array[(mid + 1 + high)/2], 0, 1);
        // high path
        __builtin_prefetch (&array[(low + mid - 1)/2], 0, 1);
#endif

        if(array[mid] < key)
            low = mid + 1;
        else if(array[mid] == key)
             return mid;
        else if(array[mid] > key)
             high = mid-1;
        }
}
```

Courtesy: Low-Level Programming: C, Assembly, and Program Execution on Intel® 64 Architecture, Igor Zhirkov

Binary search algorithm runs through the array non-sequentially and hardware cache prefetcher faces with a lot of cache misses. Int the binary search algorithm from above, we prefetch both new `low` and new `high` before we figured out which of those two values we will need. And when we actually need the data, it is already present in the cache.

The downside in this particular algorithm is that we are fetching two values, and one of those values we will never be used. This can have effect on the performance since some cache lines now need to leave the cache to make place for unused value; if another program is running on another core that uses the same cache, it could lead to performance degradation for both programs.

Like what you are reading? Follow us on LinkedIn or Twitter and get notified as soon as new content becomes available.

Experiments

Let's see how these tips help us on our real-world problems. For testing we are using a regular general purpose system AMD A8-4500M CPU with four cores, 16KB of L1 data cache per core and 2MB of L2 data cache per core (two cores share 2MB of L2 data cache). This system has 12GB of memory.

Matrix multiplication

There is an excellent but a lengthy article written by Ulrich Drepper on cache optimizations available here. He starts with naïve version of matrix multiplication algorithm and refines it until he comes to a optimized version that is almost ten times faster. In his experiment, he used a 1000×1000 matrix consisting of doubles. First optimization he did was transposing the matrix before multiplication, this alone has brought performance increase of 76.6%! A number worth noticing.

Binary Search Algorithm with Software Prefetch

We implemented a binary search algorithm that uses software prefetching from the previous chapter and we used to test it how the memory cache subsystem works when the programmer explicitly uses prefetching. The source code of the program we are using to test is available at github. To run it, simply execute `make binary_search_runtimes` and `make binary_search_cache_misses`.

We generated a sorted random integer array of length 10K, 100K, 1M, 10M and 100M elements that we use to perform binary search on. We call this `input_array` and the length of the array is our *working set*. Next, we generate another array that holds the indexes we use for searching. We call this `index_array`. We use several different approaches to generate the indexes for the `index_array`. We use a random approach to generate elements for the `index_array`, but we also use a non-random stride based approach. If the stride is N, first element of the `index_array` is 0, seconds is N, third is 2N etc, until we reach the end of the array and wrap around.

So calls to binary search looks like this:

```
1.  for (int i = 0; i < len; i++) {
2.     binarySearch(inputArray, len, inputArray[indexArray[i]]);
3.  }
```

We tested for both random `index_array` and `index_array` with strides 1, 100 and 10K. We fixed the size of the `index_array` at 10M which means we are performing 10M searches. For `index_array` with random access, these are the results:

Working Set	Prefetching Off	Prefetching On	Speed Difference
10K	1673ms	1777ms	-6.2%
100K	2478ms	2426ms	+2.1%
1M	4519ms	3996ms	+11.6%
10M	8804ms	7096ms	+19.4%
100M	14970ms	11685ms	+21.9%

Binary Search Running Time (ms) vs Working Set Size, Random Access

For random access, software prefetching is slower only for the smallest data set of 10K. As the working set increases, so does the algorithm with software prefetching becomes faster that the common algorithm. Speed difference is about 20% for large working sets, and it will probably remain that way even if the working set size increases further.

What happens if we are not accessing random elements, but instead we are accessing elements with constant stride? Since this is a synthetic test, we are looking for an element whose position we know in advance. E.g. if the stride is 100, we know the position of the first element which is 0 and we are looking for value `input_array[0]`. In the next iteration we are looking for an element `input_array[100]` etc. How does the cache behave in that case? Here are the results:

Working Set	Prefetching Off	Prefetching On	Speed Difference
10K	977ms	1168ms	-19.5%
100K	1122ms	1380ms	-23.0%
1M	1367ms	1623ms	-18.7%
10M	1610ms	1892ms	-17.5%
100M	1813ms	2171ms	-19.7%

Binary Search Running Time (ms) vs Working Set Size, Stride = 1

Working Set	Prefetching Off	Prefetching On	Speed Difference
10K	1112ms	1418ms	-27.5%
100K	1766ms	1942ms	-9.7%
1M	3018ms	2792ms	+7.5%
10M	3236ms	3019ms	+6.7%

100M	3356ms	3182ms	+5.2%

Binary Search Running Time (ms) vs Working Set Size, Stride = 100

Working Set	Prefetching Off	Prefetching On	Speed Difference
10K	760ms	984ms	-29.5%
100K	1049ms	1508ms	-43.8%
1M	2739ms	2640ms	+3.6%
10M	4402ms	7490ms	-70.1%
100M	10395ms	8251ms	-20.6%

Binary Search Running Time (ms) vs Working Set Size, Stride = 10K

For stride = 1, the common algorithm beats the software prefetching algorithm every time. This doesn't surprise however because all the data from the previous iteration is already in cache, and since the stride is 1, the current iteration has almost all the data in cache.

For stride = 100, as expected, for small working set the general algorithm beats the software prefetching algorithm. But as the working set increases, the software prefetching algorithm takes over and is on average 6% faster. Why is in this case faster on average 6% compared to 20% in completely random case? If our working set is 10M, on average the algorithm finishes in around 20 steps. Out of these 20 steps, data will already be in the cache 14 of the steps from the previous search. So, only in the last 6 steps will software prefetching make sense.

For stride = 10K we see a very weird behavior that the general algorithm is faster than the software prefetching algorithm. Why? The answer is that with a 10K stride we reduce the working set size by 10K. For the input working set of 10M, this reduces it to

a working set to only 10K. So even for the largest set size (10M) the general purpose algorithm is faster, with almost the same percentage as general purpose algorithm with random indexes and working set 10K.

How about cache performance? What kind of impact does prefetching has on cache performance. Let us compare results of perf[4] command with and without prefetching:

Parameter	Prefetching Off	Prefetching On	Difference
Cache References	409M	649M	+58.7%
Cache Misses	155M	254M	+63.9%
Cycles	31481M	25648M	-18.5%
Instructions	6659M	10647M	+59.9%

Cache Performance, Cycles and Instructions Data for Two Versions of Binary Search

The results are interesting. The software prefetching version has more cache references, more cache misses and more instructions executed compared to a regular version. What does this mean is that the software prefetching version actually does more work. But it is nevertheless faster, because it does more work in smaller number of cycles. Modern processor can execute more than one instruction per cycle, but this number falls down if the processor needs to wait for the data to be fetched from the main memory. If we count the number of instructions per cycle, it is 0.42 for software prefetching version and 0.21 for regular version. This is a huge difference.

Cache Friendly Linked List

Seconds experiment is a cache friendly linked list. Regular linked lists are very cache unfriendly, iterating through such a structure results in many cache misses. How can we do better?

Regular linked list typically consists of linked list nodes, and each node holds a value and pointer to the next node. In our implementation, linked list node can hold more than one value and the number of values is specified as a template parameter to the `linked_list` class. We tested for linked list nodes that can hold 1, 2, 4 and 8 values, with the expectation that the bigger the number of values in the node, there will be less cache misses. The source code of the program we are using to test is available at github. To run it, simply execute `make linked_list_runtimes` and `make linked_list_cache_misses`.

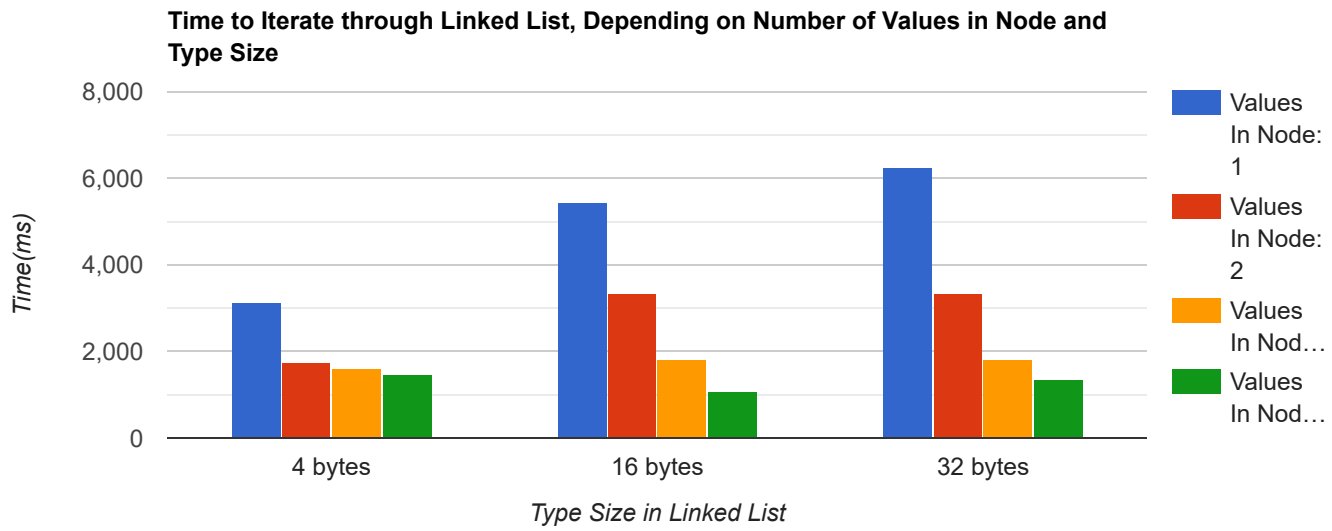
The only thing we are interested in testing is iterating through the list. For insertion, removal etc. our implementation is faster than simple linked list as well, but most of the performance improvements comes from having less memory allocations, not better usage of data cache.

Here is the source code of a single list node:

```
1.      class linked_list_node {
2.      public:
3.          char used_elems[count];
4.          linked_list_node* next;
5.          char values[count * sizeof(Type)];
6.          ...
7.  };
```

Template constant `count` holds the number of values in a single list node and template constant `Type` is the type we are keeping in the node. Since a node has more than one value, we mark which values are actually used in array `used_elems`. Notice we are using chars to hold used values, not bools. On my machine bool takes four bytes, whereas char takes one byte. This choice does increase performance of the list, since more other data from the node will be prefetched in the cache.

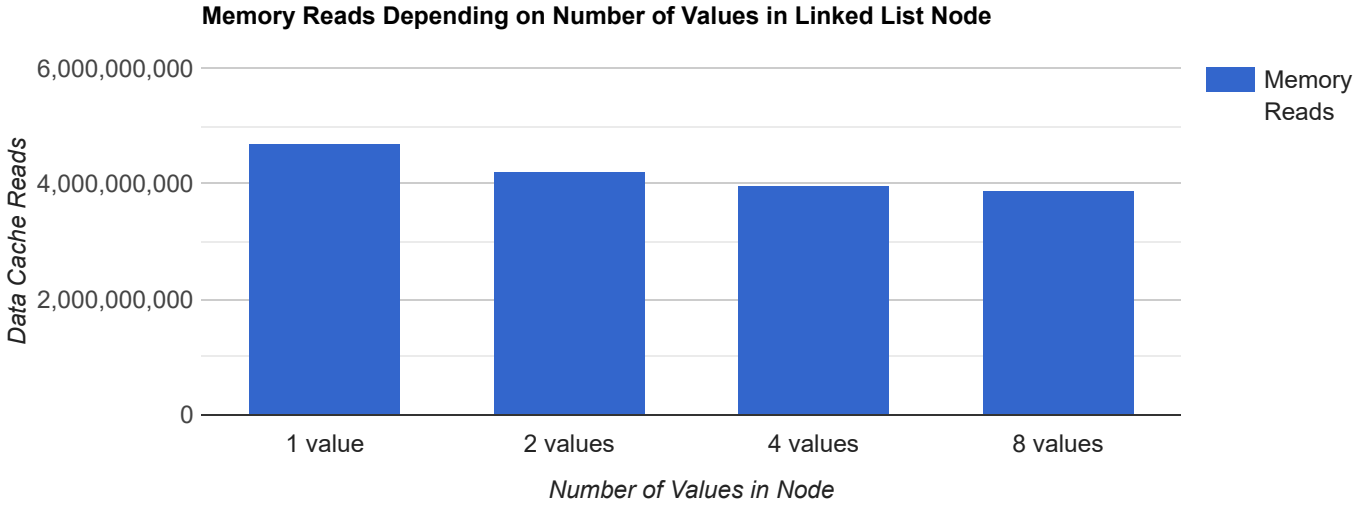
Now let's get down to measurements.[5] Bellow chart measures time needed to iterate through the linked list[6], depending on the number of values in node (1, 2, 4 and 8) and type size the linked list is holding.



The results look excellent. For smallest type size, iterating through a linked list with two values in node takes 43% less time than iterating through a linked list with one value and this number remains roughly the same for all type sizes.

Also note that the bigger the type size in the list, the more sense does it make to have more values in a single linked list node. For 4 bytes type size difference between node with two values and node with four values is small, but for 32 bytes type size this difference is significant.

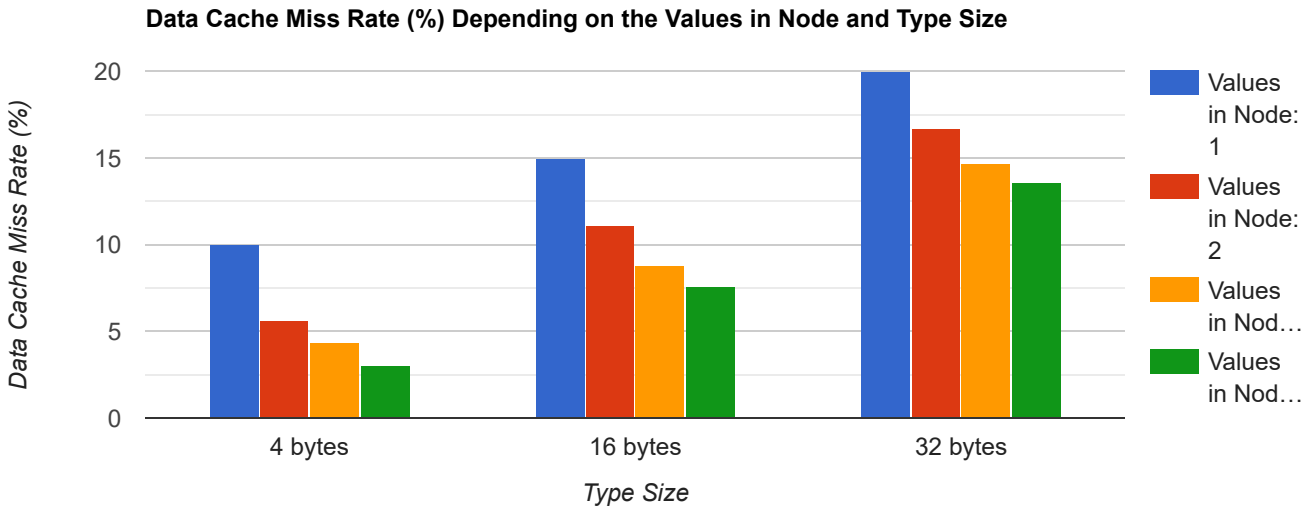
What about cache performance? What is the number of memory reads[7] and data cache miss rate for our test? First lets measure how many memory reads[8] does our program make:



As you can see, the more values stored in a single node, there will be less memory reads altogether. This is to be expected since there is less pointer arithmetic involved. Please note that

number of memory reads in our case doesn't depend on the type size.

What about cache misses? We used the tool cachegrind to measure cache miss rates[9]. It can give information about cache misses per function or per line. Here are the results:



General trend is: if there are more values in a linked list note, there will be less cache misses. For smallest type, cache rate miss is 3% for 8 values in node compared to 10% for 1 value in node. Similar ratio is observed also for larger types.

The other trend: the bigger the type size in a linked list node, the bigger the cache miss rate. For 4 bytes type size cache miss rate is 10% in worst case and 3% in best case. For 32 bytes type size cache miss rate is 20% in worst case and 13.6% in best case. Nothing unexpected. When we are accessing a first value in the node, the cache prefetcher will load values in surrounding memory addresses so when we need them later, they will already be in the cache. If the type is bigger there is less chance that the prefetcher will load useful data, so this is the reason for additional cache misses.

Final Words

So what's the conclusion? There is definitely something interesting to say about data cache optimizations, both in our

experiments and generally.

Final Words on Experiments

So what is the conclusion? Let's talk first about our experiments: all three experiments have shown that careful programming with focus on better using data cache will achieve performance gains. Optimized matrix multiplication has an excellent performance gains compared to the non-optimized version, but apart from matrices, it is rarely the case that we can do a simple manipulation on the data structure to make our structure more cache-friendly. And in my professional experience, I have rarely seen matrices used as data structures.

Second experiment was about software prefetching. We indeed see performance increase on binary search with the software prefetching, but performance increase is only relevant for working sets that do not fit the data cache. Please note that software prefetching comes with a price: our system executes more instructions and loads more data into cache which could in some cases lead to worse performance elsewhere[10]. Good thing about software prefetching is that it is easy to implement and it can be used to speed up iterating through all kinds of structures: linked lists, trees, heaps etc. Yet, careful measurement is needed to make sure that performance gains are worth it.

And our third experiment was about cache friendly linked lists. Again we see with a nice speed improvements. Improvements come from several sources: less memory allocation, smaller data cache miss rate and less instructions needed for pointer arithmetic. The only drawback are more complicated implementation and an increased memory consumption since it might happen that a single node is not optimally filled. But generally, I think these drawbacks are worth the effort.

Final Words on Data Cache Optimizations

As our experiments have shown, and from my earlier experience, the optimizations indeed bring the performance speeds, and there are numerous applications where this is needed. Performance increase depend on many factors and can range from few percent to 100% or even more.

However, some of the recommendation made here will make your program more difficult to understand and more difficult to debug. And developers have been thought to avoid exactly this, for good reasons. In order to maximally benefit from these optimizations, you will need to carefully profile your program to find the bottlenecks and focus on speeding up those, instead of applying these tips everywhere in your code. Keep your interfaces clean and avoid intermodule dependencies first, then replacing your slow code with a faster one will be simple and you can enjoy the performance increase without sacrificing maintainability.

> *Like what you are reading? Follow us on LinkedIn or Twitter and get notified as soon as new content becomes available.*

Featured image courtesy of: https://www.androidauthority.com/what-is-cache-memory-gary-explains-681747/

Further Read

Ulrich Drepper “What every programmer should know about memory”

What is cache memory – Gary explains

Agner.org – Software Optimization Resources

1. Modern day multiprocessor systems have complicated hierarchy of cache memories that is beyond the scope of this article [↩]
2. There are also other ways to avoid padding, but this is the simplest. [↩]
3. It is possible to write directly to memory without going through data cache, e.g. most implementations of memcpy do that. But it is very rare we want to do that in our programs [↩]

4. Perf is a Linux' profiler, an excellent tool I will write about [↩]
5. You can execute the tests on your system as well, just download the source code and run `make linked_list_runtimes` to execute the tests. [↩]
6. We perform 30 iterations and then measure the average [↩]
7. Memory reads and data cache reads are the same thing because all reads go through the cache [↩]
8. We measure only memory data reads since iterating through a list doesn't involve memory writes [↩]
9. In our example there were almost no LL cache misses so we measured only L1 cache misses [↩]
10. Example if another program is running on the core that shares the same L1 cache [↩]

Tagged: c c++ data cache embedded systems general purpose systems high-performance systems low-level optimizations performance

← A story about error recovery a.k.a those boring recurring bugs and what to do about them

Link Time Optimizations: New Way to Do Compiler Optimizations →

Leave a Reply

*Your email address will not be published. Required fields are marked **

Comment

Name *

Email *

Website

Save my name, email, and website in this browser for the next time I comment.

Post Comment

Search ...

Search

Recent Posts

- Flexibility and Performance
- Speedscope: visualize what your program is doing and where it is spending time
- Speeding up an Image Processing Algorithm
- The true price of virtual functions in C++
- 2-minute read: Class Size, Member Layout and Speed

Recent Comments

- Roi Barkan on Process polymorphic

classes in lightning speed
- Ivica Bogosavljević on Process polymorphic classes in lightning speed
- Roi Barkan on Process polymorphic classes in lightning speed
- Ivica Bogosavljević on Process polymorphic classes in lightning speed
- Roi Barkan on Process polymorphic classes in lightning speed

Archives

- March 2021
- February 2021
- January 2021
- December 2020
- November 2020
- October 2020
- September 2020
- August 2020
- July 2020
- June 2020
- May 2020

Categories

- Debugging
- Developer Tools
- Memory Footprint
- Performance
- Reliability

Meta

- Log in

- Entries feed
- Comments feed
- WordPress.org