Software Optimization: Fixing Memory Problems

Abstract

Improving memory performance is of paramount importance when attempting to optimize software speed. Luckily, there are a number of techniques that can be used to increase memory efficiency. This article examines a variety these techniques for improving memory performance, including resolving important store forwarding problems, reducing compulsory cache misses, increasing cache efficiency, using hardware or software prefetch, utilizing non-temporal instructions, avoiding buffer conflicts, avoiding capacity issues, and moving nondata dependent work to the locations of cache misses.

Introduction

An important skill for developers is the ability to recognize performance issues by using performance analyzers, running performance experiments, and appraising code directly. In the process of learning that skill, developers must become familiar with the common pitfalls affecting memory performance and learn to recognize and resolve these issues when creating software.

Fixing Memory Problems

Once the details of where and why a memory access is a bottleneck, optimizations can be made. The following list describes the techniques used to improve memory performance.

- **Fix any store forwarding issues that are found.** This effort usually is as easy as performing the necessary data accesses using the same type as was used to store the value.
- **Use less memory to reduce compulsory cache misses.** Selecting a different algorithm that uses less memory can help. For example, some sorting algorithms like insertion sort operate on the array in-place while other sorting algorithms like merge sort require additional temporary memory. Make sure to select a computationally and memory efficient algorithm. Remember, computationally efficient algorithms can make a much bigger difference than a few extra cache misses so don't automatically select an algorithm based solely on its use of memory. Changing data types can also reduce the amount of memory. If 32-bit integers are not needed, try words or bytes or even bits. This recommendation applies to Intel EM64T applications in particular since pointers are 64 bits. If instead you can change some pointers to 32-bit indexes into arrays, you would save 32 bits per pointer, and this savings can often make the difference between an application that fits well into caches, and one that doesn't. Sampling on 1st Level Cache Misses Retired event shows the locations where the application is using memory. Use this as a guide to steer you towards the areas of your application where using less memory would have the

biggest impact. Also, you should consider sampling on L2 cache misses since the biggest jump in latency occurs between L2 and main memory. If L2 cache misses are quite low, reducing the memory footprint of the program is unlikely to significantly increase performance.

- **Increase cache efficiency.** Examine what cache lines are being loaded to make sure that all the memory is being used. Adjust data structures and memory buffers to place items used at the same time next to each other in memory. No tools exist today that show cache efficiency. But, you can get an idea of the cache efficiency by seeing how many cache lines have been loaded versus the amount of memory you expected your application to use.

- **Read memory sooner with prefetch.** Try to arrange your data structure accesses so that hardware prefetch naturally prefetches the data. An example might be if a pointer-chasing loop takes a lot of time, try to arrange the pointers in the list so that they are naturally strided in a way that often causes the hardware prefetcher to fetch these addresses. If the data cannot be arranged to allow hardware prefetch, the software prefetch instructions can be used to bring memory into the caches earlier. By issuing the prefetch instruction far enough before the data is needed, a cache miss still occurs, but now the data waits in the cache for the processor instead of the processor waiting for the data. Be sure that you always use the benchmark to test that the use of software prefetch instructions improves performance because sometimes the use of software prefetch instructions can cause performance degradation.

- **Write memory faster with non-temporal instructions.** The streaming, non-temporal instructions write data without using the cache, saving one cache read, caused by the read-for-ownership cache policy, and one cache write. When using the non-temporal instructions, make sure that the data is not loaded in the near future by another function. Writing to memory with the nontemporal instructions just to read it back into the cache does not improve overall performance. The non-temporal instructions work best when writing data that is never used again by the processor, such as frame buffer data that is used only by the graphics card.

- **Avoid conflicts.** The address of the data being accessed determines where in the cache it can be placed. Avoid reading or writing nine or more buffers at the same time with the same 2-kilobyte alignment or the L1 cache has to evict a cache line even if the cache is not full. Detecting the 2-kilobyte L1 cache alignment conflicts is rather difficult because the processor does not have an event counter that tracks this situation. A combination of looking at the suspected source code, inspecting the accessed data addresses in a debugger, and running performance experiments can be used to identify L1 cache conflicts. When developing a performance experiment to detect L1 cache conflicts, force the addresses to be aligned differently or stop accessing a memory buffer or two. If L1 cache conflicts were occurring, you should be able to detect a change in the number of L1 cache misses using the VTune analyzer. The 64-kilobyte or 4-megabyte cache controller conflicts can be detected by sampling on the 64K Aliasing Conflicts processor event counter with the VTune analyzer.

- **Avoid capacity issues.** Capacity issues are caused by the eviction of data before all references to it are finished. This problem usually occurs in two-pass algorithms where a large buffer is processed by one function, followed by a second-pass over the same buffer by a second function. Both functions cause cache misses even though the same data is used. Instead, try operating on smaller cache sized buffers. So, run the first function on a cache-sized subset of data, run the second function on the same cache-sized subset of data. If everything goes as planned, the second function would have no cache misses because the data is still in the cache from when the first function loaded it. Then, repeat both the first and second functions on the next cachesized subset of data, and so on.

- **Be careful** when determining a cache-sized subset of data to operate on because all memory accesses, including stack variables, global variables, and the buffers, all contribute to capacity issues. It is very rare that the full L1 cache size (16- or 32-kilobyte) can be used because of all the other variables. Try to pick a size that is easy to program and avoids capacity issues, such as 4 or 8 kilobytes of memory. Use the 1st Level Cache Misses Retired event counter to find the locations of the cache misses events. Then, by just looking at the source code, determine whether the memory was just in the cache.

- **Add more work.** The processor can execute non-dependent instructions while waiting for memory to be fetched. Where possible, take advantage of these "free" clocks by moving nondata dependent work to the locations of cache misses. The cache misses still take the same amount of time, but now the processor can execute other instructions during the wait instead of just wasting time.

Example 1.1 Optimize a Function

An important optimization skill is being able to look at a piece of code and predict the performance issues and solutions without using a performance analyzer. Look at the following loop and determine what issues exist.

Problem

Improve the following function assuming that the Dest array would not be used in the near future, that the arrays are aligned, and that len is a multiple of four.

```
void AddKtoArray (int Dest [], int Src[], int len, int K)
{
        int i;
        for (i=0; i<len; i++)
        Dest [i] = Src [i] + K;
}
```

Solution

The first thing to notice is that this loop becomes memory bound when the arrays are not already in the cache because it does nothing else that is time-consuming. Since the problem statement says that the destination array is not be used in the near future, the streaming store instructions should be used.

The best way to improve this loop is to use the streaming store instructions and the SIMD instructions to add four integers at a time.

These changes improve performance by about 33 percent, but the loop is still very memory bound. Further improvements can be made by adding more work to this loop using the time that would be wasted waiting for memory, by reducing the amount of memory used which also reduces the number of cache misses, or by making sure that the memory was in the cache using a strip-mining technique with another function that accessed the same memory. The following code segment uses the intrinsics and the Intel C++ Compiler's class libraries.

```
// assumes: arrays are 16 byte aligned
// len is a multiple of 4
void AddKtoArray4s (int Dest [], int Src[], int len, int K)
{
        int i;
        __m128i *Dest4 = (__m128i *)Dest;
        Is32vec4 *Src4 = (Is32vec4 *)Src;
        Is32vec4 K4(K, K, K, K);
        for (i=0; i<len/4; i++)
        _mm_stream_si128(Dest4+i, Src4[i] + K4);
}
```

Example 1.2 Optimize a Data Structure

Looking at data structures and immediately identifying cache issues is an important part of software optimization. Look at the following data structure and identify possible performance issues and improvements.

Problem

Optimize a phone book data structure to improve searching. The data structure is:

```
#define MAX_LAST_NAME_SIZE 16
typedef struct _TAGPHONE_BOOK_ENTRY {
        char LastName[MAX_LAST_NAME_SIZE];
        char FirstName[16];
        char email[16];
        char phone[10];
        char cell[10];
        char addr1[16];
        char addr2[16];
        char city[16];
        char state[2];
        char zip[5];
        _TAGPHONE_BOOK_ENTRY *pNext;
} PhoneBook;
```

Intel SWBC2; 0-9764832-1-1; Production Proof; 01-30-06 #1

140 The Software Optimization Cookbook

The search function is:

```
PhoneBook * FindName(char Last[], PhoneBook * pHead)
{
        while (pHead != NULL)
        {
                if (stricmp(Last, pHead->LastName) == 0)
                return pHead;
                pHead = pHead->pNext;
        }
return NULL;
}
```

Solution

First, recognize that the problem is that the function makes horrible use of the cache. Each structure takes up 127 bytes, but only 20 bytes are used for each pass through the search loop. This arrangement wastes 48 bytes of the 64-byte L1 cache line and 111 bytes of the 128-byte L2 cache line meaning the L1 cache efficiency is at best 25 percent. To improve performance, rearrange the structure so that all the last name variables are in one continuous array and place the other less frequently used data somewhere else. The two arrays would be declared as shown in the following sample.

```
char LastNames[MAX_ENTRIES * MAX_LAST_NAME_SIZE];
PhoneBook PhoneBookHead[MAX_ENTRIES];
```

Since last names can be any length, up to MAX_LAST_NAME_SIZE-1 bytes still might be wasted in the array, but you have made a big improvement over the previous version. Taking into consideration variable length strings, the find function can now be written:

```
PhoneBook * FindName(char Last[], char *pNamesHead,
PhoneBook *pDataHead, int NumEntries)
{
        int i = 0;
        while (i < NumEntries)
```

```
            {
                    if (stricmp(Last, pNamesHead) == 0)
                    return (pDataHead+i);
                    i++;
                    pNamesHead += strlen(pNamesHead) + 1;
            }
            return NULL;
    }
```

The bottleneck in this code shifts to the string compare and string length functions, which should now be optimized with application specific versions. See Chapter 10, “Slow Operations,” for additional information.

A further improvement would be to replace the sequential search algorithm with a binary search or other higher-performance algorithm.

Conclusion

Once any memory issues are detected, one can use a variety of techniques to improve memory performance. One technique is to resolve any important store forwarding problems, which can usually be done by matching the stores and load addresses. Another technique is to use less memory in order to reduce compulsory cache misses, perhaps by choosing a different algorithm that is more computationally and memory efficient or by changing data types to words or bytes or bits. Cache efficiency can be increased by adjusting data structures and memory buffers so that items used at the same time are placed next to each other in memory. In addition, prefetch can be used to read memory sooner; data structure accesses can be arranged so that hardware prefetch naturally prefetches the data, or software prefetch instructions can be used to bring memory into the caches earlier. Non-temporal instructions for writing data that is never used again by the processor can increase write memory efficiency. Also, to prevent a cache line from being evicted by the L1 cache, reading or writing nine or more buffers at the same time with the same 2-kilobyte alignment should be avoided. To avoid capacity issues caused by the eviction of data before all references to it are finished, one can try operating on smaller cache sized buffers. Finally, moving nondata dependent work to the locations of cache misses allows the processor to execute non-dependent instructions while waiting for memory to be fetched. For further reading on the subject, t*he Software Optimization Cookbook, Second Edition* by Gerber et al. contains chapters on detecting memory issues and diagnosing memory performance problems.

This article is based on material found in the book *The Software Optimization Cookbook, Second Edition*, by Richard Gerber, Aart J.C. Bik, Kevin B. Smith, and Xinmin Tian. Visit the Intel Press website to learn more about this book:

http://noggin.intel.com/intelpress/categories/books/software-optimization-cookbook-second-edition

Also see our Recommended Reading List for similar topics:

http://noggin.intel.com/rr

About the Authors

Richard Gerber has worked on numerous multimedia projects, 3D libraries, and computer games for Intel. As a software engineer, he worked on the Intel VTune™ Performance Analyzer and led training sessions on optimization techniques. He is the author of The Software Optimization Cookbook.

Aart J.C. Bik holds a PhD in computer science and is a Principal Engineer at Intel Corporation, working on the development of high performance Intel® C++ and Fortran compilers. Aart received an Intel Achievement Award, the company's highest award, for making the Intel Streaming SIMD Extensions easier to use through automatic vectorization. Aart is the author of The Software Vectorization Handbook.

Kevin B. Smith is a software architect for Intel's C and FORTRAN compilers. Since 1981 he has worked on optimizing compilers for Intel 8086, 80186, i960®, Pentium®, Pentium Pro, Pentium III, Pentium 4, and Pentium M processors.

Xinmin Tian holds a PhD in computer science and leads an Intel development group working on exploiting thread-level parallelism in high-performance Intel® C++ and Fortran compilers for Intel Itanium®, IA-32, Intel® EM64T, and multi-core architectures.