8 Steps to Optimizing Cache Memory Access and Application Performance
A recommended approach for cache memory optimization

A White Paper by Rogue Wave Software.

September 2011
8 Steps to Optimizing Cache Memory Access and Application Performance

A recommended approach for cache memory optimization

by Rogue Wave Software

© 2011 by Rogue Wave Software. All Rights Reserved

Printed in the United States of America

Publishing History:

September 2011

Trademark Information

The Rogue Wave Software name and logo are registered trademarks of Rogue Wave Software, Inc. or its subsidiaries in the US and other countries. ThreadSpotter is a trademark of Rogue Wave Software, Inc. or its subsidiaries. All other company, product or brand names are the property of their respective owners.

IMPORTANT NOTICE: The information contained in this document is subject to change without notice. Rogue Wave Software, Inc. makes no warranty of any kind with regards to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Rogue Wave Software, Inc. shall not be liable for errors contained herein or for incidental, consequential, or other indirect damages in connection with the furnishing, performance, or use of this material.
# TABLE OF CONTENTS

8 Steps to Optimizing Cache Memory Access and Application Performance ........................................... 2
Initial State: A Correct Program and a Good Test Case ............................................................................. 5
Step 1: Start with High-Level Optimizations ............................................................................................... 7
Step 2: Optimize Data Layout ....................................................................................................................... 7
Step 3: Optimize Access Patterns .................................................................................................................. 8
Step 4: Utilize Reuse Opportunities ............................................................................................................ 9
Step 5: Use Non-Temporal Hints for Data without Temporal Reuse ......................................................... 10
Step 6: Minimize Communication between Caches ..................................................................................... 11
Step 7: Avoid False Sharing ......................................................................................................................... 11
Step 8: Hide Remaining Misses ................................................................................................................... 12
Summary ....................................................................................................................................................... 12
About ThreadSpotter Multicore Analysis Tool ........................................................................................... 13
About Rogue Wave Software ....................................................................................................................... 13
Optimizing an application for good cache performance involves distinct steps, each targeting a different category of problems. The order of the steps is somewhat important as some problems obscure others, and some optimizations enable other optimizations or make other optimizations unnecessary. By following this workflow you avoid making unnecessary optimizations and missing optimization opportunities that may not present themselves clearly if the steps are done in another order.

The workflow initial state is a correct program and a good test case; the flow moves through addressing data layout and data access patterns, exploring data reuse opportunities, considering multithreading issues, and finally, putting on the finishing touches. This flow is outlined in the figure below.

Figure 1. Steps to optimizing cache performance.
Initial State: A Correct Program and a Good Test Case

First, make sure the program is correct. If it is not, and must be changed after it has been optimized, some optimizations may not have been required, or may not work as intended with the corrected program and must be undone. Therefore, avoid unnecessary work by making sure the program is correct before you start to optimize it.

Second, create a good test case, one that satisfies these criteria:

- **Measurable and repeatable performance**
  The test case will be used to check how performance changes when the program is changed, so it must produce a clear measure of performance that can be compared between different versions of the program.

  The simplest programs to measure are those that perform a fixed amount of processing based on their input. Simply measure how long the program takes to run—the faster the better.

  More challenging to measure are programs that run continuously, processing work as it arrives, such as server applications. The performance of these programs is usually measured as throughput, in number of transactions per second, bytes per second, or something similar. You must set up a repeatable load and a way to measure either how long it takes to process a certain amount of work or how much work is processed during a certain amount of time.

  The test case must also give repeatable performance measurements. Run multiple times, the test case should produce near identical performance figures with each run. If the performance varies between runs it will be difficult to determine if a change in performance is caused by a change in the program or is just a run-to-run variation.

- **Relevant and full-size**
  The test case must stress the relevant parts of the code and run a full-size workload. The best way to ensure this is to use a real workload.

  The advice ThreadSpotter provides is based on how the data set interacts with the memory cache hierarchy of the processors. If you use a small, test workload, the data set for the workload may fit in the caches, even though the data set of a full-size workload would not. With a small workload, therefore, ThreadSpotter would not find problems that would occur with a full-size workload.
If running a full-size workload takes too long to be practical, it is most likely better to just run a part of the execution of a full-size workload than to use a reduced workload to shorten the execution time.

- **Verifiable correctness**
  This is not strictly necessary, but if possible the test case should produce output that can be verified for correctness. This makes it easy to establish that each change to the program does not break it while you test whether the change improves performance.

- **Insensitivity to execution speed**
  Sampling a program with ThreadSpotter introduces a certain performance overhead. If possible, therefore, you should make sure that the work that the program performs is not dependent on the execution speed.

  For example, a networking application may discard incoming network packets if they arrive at a higher rate than that at which it can process them. If this starts to happen because of the sampling overhead then the sampled execution will not be identical to a normal execution and therefore not fully representative.

  If the execution changes depending on the execution speed it also makes it harder to compare the program behavior when changing the code.

There are several ways to compare the cache behavior before and after a code change. You could compare miss and fetch ratios, but if a code change alters the number of accesses as well as the number of misses, the miss ratio will not be a reliable indicator of improvement. With a repeatable test case, it makes more sense to compare the absolute number of misses before and after the algorithm change. If the test case is not absolutely repeatable, it will be difficult to correlate two measurements of the absolute number of misses or fetches.

**Tips**

- Ensure that the program is correct.
- Create a good test case.
- Take a reference measurement of the execution time.
Step 1: Start with High-Level Optimizations

Generally speaking, the greatest performance improvements come from algorithm improvements. If your program does a bubble sort, you will see more improvement from switching to the quick sort algorithm than from optimizing the bubble sort. Therefore, start by ensuring that your application uses efficient algorithms.

Make sure that you use the best available compiler and the correct compiler flags. Switching compilers or changing compilation flags may improve performance with very little work.

Make sure that your program does not do unnecessary work. Use an execution profiler (such as gprof or callgrind) to find hot-spots in the code. These profilers measure the amount of time spent in different parts of the program and direct your attention to highly-used code sections. Profiling an application allows you to evaluate issues like:

- Is the program doing unnecessary work, such as unnecessarily repeating an expensive operation? Try to reuse intermediate results that are expensive to calculate.
- Does the program contain unnecessary copying of data structures? This often occurs with programming paradigms that advocate layering, encapsulation and data hiding. Consider relaxing encapsulation rules where that offers performance improvements.

**Tips**

- Use efficient algorithms.
- Use optimizing compilers and turn on suitable optimization flags.
- Avoid unnecessary work by reusing intermediate results that are expensive to calculate.
- Judiciously break encapsulation to avoid copying data.

Step 2: Optimize Data Layout

Once the program is algorithmically sound you can start looking at memory-related optimization areas. First, ensure that data is optimally laid out in memory.

Generally speaking, data should be laid out as compactly as possible to utilize the cache space and memory bandwidth well. It should also be laid out in such a way that as much as possible of the data in each cache line loaded into the cache is used; data being used in the same code context should be placed contiguously in memory.

ThreadSpotter will point you to places in the code that are using sparse data structures and places where data structures are only partially used.
Here are some situations that can cause unused space in data structures and inefficient use of cache space and memory bandwidth:

- **Internal alignment gaps**: the compiler aligns the offsets of fields in structures depending on the alignment requirements of their data types. This can cause consecutive fields in a structure to be allocated with unused space between them. Sorting fields by decreasing alignment requirements will minimize this waste.
- **External alignment gaps**: alignment gaps can also occur between structures in an array due to alignment requirements for structures. If possible, make sure the structure size is a multiple of the required alignment.
- **Unnecessarily large data types**: use the smallest possible data types to minimize the amount of cache space and memory bandwidth used.
- **Mixing frequently-used fields with rarely-used fields in structures**: it may be better to split the structure into two, one for the frequently-used fields and one for the rarely-used fields.
- **Mixing read-only fields and written fields in structures**: even if only one field in a cache line is written, the whole cache line, including all the unchanged data, has to be written back to memory, wasting memory bandwidth.
- **Dynamic memory allocation of small structures**: the memory allocator’s housekeeping data between allocated structures are used much less frequently than the allocated structures.

**Tips**

- Use the smallest possible data types.
- Organize data to avoid mixing frequently used and rarely used fields.
- Organize data to avoid mixing read-only and written fields.
- Sort fields in structures according to alignment requirements.
- Avoid dynamic allocation of small structures.

**Step 3: Optimize Access Patterns**

Inefficient access patterns are access patterns that tend to use only a small part of each accessed cache line and not reuse the data in the line before it is evicted from the cache.

One common form of inefficient access patterns is irregular access patterns. They occur if the data set is not allocated in memory in the same order as the accesses occur. For example, following a linked list may generate irregular accesses if the list elements have been spread out over the heap by the memory allocator.

Certain high level data structures such as hash tables and tree structures will cause irregular accesses patterns by the way they are designed. When the accesses to
such a structure become a problem from a cache usage viewpoint it is necessary to test whether using a more cache-friendly data structure could improve performance on a case-by-case basis.

Another kind of inefficient access pattern occurs when a matrix data structure is being traversed along the wrong dimension, across the cache lines instead of along the cache lines. This is also known as inefficient loop nesting.

ThreadSpotter will pinpoint locations in the source code causing inefficient memory access patterns and classify the cause of the inefficiency.

**Tips**

- Organize data or change data access order so that the application accesses contiguous locations in memory.
- Avoid data structures that are irregular in nature, or cause irregular access patterns.
- For data structures that involve following pointers, such as linked lists and trees, consider allocating memory for chunks of nearby elements together to make the accesses more regular.
- Sometimes creating a local data structure optimized for the current algorithm can improve performance. For example, copying the elements in a tree data structure to a local sorted array may improve performance if they are only going to be accessed in that order.

**Step 4: Utilize Reuse Opportunities**

Once a cache line has been fetched into the cache you want to use the data in it as much as possible before it is evicted, so that the number of times each line has to be fetched into the cache is minimized. ThreadSpotter helps you do this by pointing out and classifying unexploited data reuse opportunities in the code.

Basically there are two common types of unused data reuse opportunities: reuse within a loop and reuse between different loops. The former is addressed by applying a technique called blocking or tiling, and the latter is handled by bringing the two loops closer together or even merging the loops if possible.

Reuse of cache lines comes in two distinct flavors: spatial reuse, where the accessed data is close to other data that has already been fetched into the cache, and temporal reuse, where the same data is revisited multiple times. Minimizing the time before the reuse will lower miss ratios.
Addressing these issues will usually cause loop hierarchies to be partially turned around and broken up. Loop fusion, where two or more loops are merged into one, will also change the structure of the program. This changed structure should be reanalyzed for poor access patterns and further data reuse opportunities.

### Tips

- Bring loops accessing the same data closer, or merge them if possible.
- Look for spatial reuse and change loop nesting or block with respect to the data structure that displays long reuse distance.
- Look for temporal reuse and change algorithms to perform as many iterations as possible for a given data chunk before moving along.
- After merging loop bodies or transforming loop structures, rerun analysis to get advice for the new program structure.

### Step 5: Use Non-Temporal Hints for Data without Temporal Reuse

Sometimes it is simply not possible to reuse a cache line before it will be evicted from the cache. Most modern processors have instructions that can be used by programs to optimize cache usage in such cases.

Use non-temporal prefetch instructions to hint to the processor that you know the cache lines will be evicted from the cache before they are reused. This allows the processor to free up the cache space used for the data, which may allow other data that would otherwise have been evicted to be successfully cached.

When writing contiguous regions of non-temporal data, use non-temporal store instructions instead of non-temporal prefetches to avoid fetching the overwritten data from memory.

ThreadSpotter will point out places in the code where non-temporal prefetches and non-temporal stores should be used.

### Tips

- Give the processor a hint about cache lines you know will not be reused using non-temporal prefetch instructions.
- Write contiguous regions of non-temporal data using non-temporal store instructions.
Step 6: Minimize Communication between Caches

Analogous to optimizing data layout for efficient cache and memory bandwidth usage, arranging data for efficient cache-to-cache communication is important for multithreaded application performance. Whenever one thread uses data that another thread has written, the written data has to be communicated between the caches. As with the memory communication, such communication is done in cache line-sized chunks of data.

The first thing to do is to try to completely avoid as much of the communication as possible. However, unless the program is very simple there will be some communication between threads that is unavoidable.

For communication that cannot be avoided, try to write as large contiguous regions of memory as possible before letting other threads read the data. In communicated data structures, avoid mixing fields that are written by the producer thread with fields that are not. This minimizes the number of cache lines that have to be communicated between the caches, and therefore the effect of the communication on performance.

ThreadSpotter will point out locations in the code that cause a lot of cache-to-cache communication, both where the data is written and where it is being read. ThreadSpotter will also point out locations where partially written cache lines are causing inefficient communication.

**Tips**

- Avoid unnecessary communication.
- Write at least a cache line’s worth of data to a memory region before letting the consumer thread start reading the data.

Step 7: Avoid False Sharing

In the presence of multiple threads a number of sharing effects may affect performance. One such sharing pattern is known as false sharing. It arises if multiple threads access and update unrelated data in the same cache line. Even though the threads do not actually communicate, the cache line still has to be communicated back and forth between the threads’ caches with all the overhead that that causes.

You can avoid false sharing by making sure that unrelated data accessed by multiple threads is not allocated to the same cache line, for example, by giving explicit alignment pragmas to the compiler.
In OpenMP programs, false sharing occurs when several threads maintain their respective partial result in a vector indexed by the thread rank. Replacing this configuration with thread local variables often helps.

**Tips**
- Align frequently-accessed global data to cache line boundaries to avoid allocating variables that may be accessed from different threads in the same cache line.
- Do not store thread-specific data in an array indexed by the thread ID or rank.
- When parallelizing an algorithm, partition data sets along, not across, cache lines.

**Step 8: Hide Remaining Misses**

After all algorithms and data structures have been optimized, there may still be some places in the code that cause cache misses that cannot be avoided. You can sometimes further improve performance by carefully adding prefetch instructions to make sure that this data is loaded into the cache before it is actually needed by the program, so that the program does not have to stall waiting for the data to arrive from memory.

It can be difficult to determine either the correct place to insert the prefetch instruction or how far ahead in the data structure to prefetch. Sometimes it helps to augment the data structure being traversed with a field containing a hint of the address to prefetch, for instance the address of an element several steps ahead in a linked list.

If you insert a prefetch instruction and reanalyze the program with ThreadSpotter, ThreadSpotter will provide feedback on whether the prefetch instruction is doing useful work and whether it is prefetching data too close to or too far ahead of the use of the data.

**Tip**
- Look for remaining hot-spots and try inserting prefetch instructions.

**Summary**

Optimizing an application’s use of cache memory is critical in its overall performance. Today’s modern multi-core pipelined CPU architectures operate on data much faster than the data can be supplied by main system memory. If the cache is not always full of the data the CPU requires next, all of that CPU power stalls, sitting idle waiting for the next piece of work.
The workflow outlined here provides a solid framework in which developers can work to optimize performance of key parts of an application. The steps described, from optimizing general access patterns to advanced techniques to multithreading concerns, cover all of the areas where cache memory bottlenecks can occur. Developers who follow these steps will reach their goal of a faster application more quickly than those who follow ad-hoc methods where they may waste time attempting to optimize the wrong parts of their code.

About ThreadSpotter Multicore Analysis Tool

Rogue Wave Software’s ThreadSpotter is a performance optimization product for single- and multithreaded applications. ThreadSpotter helps eliminate performance issues by analyzing memory bandwidth and latency, data locality, and thread communication, to pinpoint performance issues. In addition to identifying bottlenecks, ThreadSpotter calculates and analyzes an application’s cache use, and provides guidance for achieving multicore efficiency, suggesting appropriate performance fixes. ThreadSpotter estimates each performance issue’s importance and rank order, and guides you to the location in the source code where the issues are located.

Simply parallelizing code is no guarantee for optimal performance. If you are truly interested in performance, first optimizing your code with regard to cache usage can dramatically multiply results from your subsequent parallelization efforts. Using ThreadSpotter to optimize for multicore will increase the performance and productivity of your applications, server resources, and programming team.

About Rogue Wave Software

Rogue Wave Software, Inc. is the largest independent provider of cross-platform software development tools and embedded components for the next generation of HPC applications. Rogue Wave marries High Performance Computing with High Productivity Computing to enable developers to harness the power of parallel applications and multicore computing. Rogue Wave products reduce the complexity of prototyping, developing, debugging, and optimizing multiprocessor and data-intensive applications. Rogue Wave customers are industry leaders in the Global 2000, ISVs, OEMs, government laboratories and research institutions that leverage computationally-complex and data-intensive applications to enable innovation and outperform competitors. For more information, visit http://www.roguewave.com.