Software Optimization: Detecting Memory Issues

Abstract

Memory performance is an essential component of processor efficiency, and it is thus vital to detect and eliminate memory issues that force the processor to wait for data. This article presents strategies for evaluating the application and identifying problematic locations that cause lapses in memory, including page swapping, store forwarding problems and L1 cache misses. It explains what features of the Intel VTune™ Performance Analyzer and the Microsoft Performance Monitor can be used to monitor these issues. The article also includes illustrative time-based sampling and experiments using sample code. These experiments demonstrate how to apply and interpret the results as well as how to evaluate which memory issues are worth resolving.

Introduction

Bottlenecks resulting from underperforming memory are highly detrimental to processor efficiency. It is essential to evaluate and identify the sources of memory constraint. This article will explain the common issues that affect memory and prevent it from performing at peak efficiency. It will also introduce the tools that can detect the presence and location of such obstacles. By the end of the article, developers should be able to evaluate their application and detect page swapping, store forwarding problems and cache misses. Developers should also have the ability to evaluate if the memory issues cause a loss in performance that is worth ameliorating before beginning work to eliminate the issue.

Detecting Memory Issues

Memory optimizations rely upon accurately detecting the location of and the reason for a memory problem. Memory can be a problem anytime the processor has to wait for its data, a situation often caused by things like page swapping and cache misses. Tracking down these locations then knowing what optimizations you can perform to make the biggest difference is the focus of this section.

Some insight into the application is very helpful for setting expectations. Does the application use a small or large amount of memory relative to the amount of physical memory? Do you expect to see continuous page swapping or maybe just a little during initialization?

Are data structures carefully planned to minimize cache misses or are they haphazardly thrown together?
Finding Page Misses

Page swapping is always a sign that the processor is waiting for memory and these situations should be eliminated where possible. The quickest way to detect page swapping is to sample on the Pages/sec operating system counter in the Memory performance object, using either the VTune Performance Analyzer or the Microsoft Performance Monitor (PERFMON.EXE). Figure 1.1 shows the Performance Monitor sampling on the event.

![Performance Monitor Tracking Pages per Second](image)

**Figure 1.1** The Performance Monitor Tracking Pages per Second

The graph in Figure 1.1 shows that the system is generating a large amount of page misses. Since every page miss costs a huge amount of time, it is important to focus optimizations on removing or at least minimizing the page misses. Aside from adding more memory to the computer, the only way to avoid page misses is by changing the application to use less memory or to use it differently to increase page locality, and therefore processor cache locality. It should be fairly easy to determine what part of the application is generating page misses by examining where large buffers and memory allocations are occurring. Make sure to consider calls made by the application to operating system functions or to other applications that may be causing page misses on the application’s behalf.

Page misses can be transient, meaning that the second time the application is run, different pages might be in memory and the profile could look different.
The Counter Monitor feature in the VTune analyzer can help pinpoint what code was running when page misses occurred. Figure 1.2 shows that the HUFF.EXE sample program causes a bunch of page misses, but only right at the start, which is the initialization code.

**Figure 1.2** VTune Analyzer, Counter Monitor Sampling Memory Pages per Second

Concentrating on the spike in the graph reveals that HUFF.EXE was the only active process, as shown in Figure 1.2.
With this information and specific knowledge about how the program works, the load of the uncompressed data file is very likely the cause of the page misses. Since no page misses occur after the beginning and loading the data file is unavoidable, this program does not have a page miss issue worth optimizing.

**Finding Store Forwarding Problems**

Store forwarding isn’t a frequent cause of performance problems. However, if these events are occurring frequently in a program, they can cause a big performance drop. So it is good to rule out or fix any store forwarding problems before working on the rest of the memory accesses. The easiest way to find store forwarding problems is to use the VTune analyzer on a Pentium 4 processor and to sample on the MOB Loads Replays Retired event. Unfortunately no similar event exists on the Pentium M processor, so finding store forwarding issues on these processors is more difficult, and you might choose not to check for them. Figure 1.4 shows the VTune analyzer’s sampling information for MOB Loads Replays Retired, being used to detect any store forwarding problems that might exist in the HUFF.EXE application. In this case, quite a few MOB Loads Replays Retired events are found in the function...
HuffCompress. However, deeper analysis shows that these events are store forwarding problems that are related to disambiguation between stores and load addresses, not to the size or offset mismatch store forwarding issues that are easily remedied. Additionally, the clockticks show that the function HuffCompress is not as hot as other functions in the program, so although some performance might be gained if the MOB replays could be lowered, this area doesn’t look fruitful for further work.

Finding L1 Cache Misses
Once page swapping is under control, and simple store forwarding improvements have been made or ruled out, it is time to focus on the L1 cache. Except for write combining (WC) memory stores and uncachable memory, all memory accesses go through the L1 cache. Sampling on L1 cache misses identifies the portions of the application that are accessing memory or at least missing the L1 cache. Comparing those locations to the application’s time-based hotspots shows you where the processor is waiting for memory. You should examine only the locations that consume a significant amount of time and have L1 cache misses.
On a Pentium 4 processor sampling on the 1st Level Cache Load Misses Retired counter in the VTune Performance Analyzer would show you where all the L1 cache load misses are occurring, as shown in Figure 1.5 for the HUFF.EXE application. Using the VTune analyzer on a Pentium M processor, sampling of the event L1 Lines Allocated provides the best approximation to L1 cache misses.

The graph in Figure 1.5 shows that the time-based hotspots and the location of the L1 cache misses are not correlated. The function HuffCompress contains the most L1 cache misses but does not consume the most time. And the function AppendBits consumes the most time but does not contain the most cache misses. In this case, optimization efforts would focus on the functions that consume the most time first. So, the order of optimizations would be AppendBits, GetCode, and then HuffCompress.

Since only three functions have L1 cache misses and they are relatively short, it would be worth the effort to examine the source code and list by hand all memory accesses. The goal would be to find cache misses occurring on the same buffers, low cache efficiency, or cache conflicts caused by multiple aligned pointers accessing memory.
**Understanding Potential Improvement**

Before fixing memory problems, it is important to make sure that memory really is the bottleneck. Just because sampling identifies a location of the application that contains many cache misses, it does not necessarily mean that a significant amount of performance is being lost. Due to out-of-order execution, it is hard to tell exactly how much time is lost waiting for memory accesses by only looking at the sampling data from the VTune analyzer. Performance experiments should be used to supplement the sampling analysis to determine how much performance is being lost. For example, let’s say that time-based sampling has identified a hotspot on a function that is also an L1 cache miss hotspot. This discovery is a very strong sign that memory accesses are causing the bottleneck, but it is still not a sure thing. Verification and quantification of the size of the bottleneck can be determined with performance experiments. The following sample piece of code adds a constant value to every element in an array.

```c
for (x=0; x<len; x++)
    DestArray[x] = SourceArray[x] + K;
```

This loop breaks down into the following steps:

1. Load `SourceArray[x]`.
2. Add `K`.
4. Increment `x`.
5. Compare `x` and `len`.
6. Jump to step 1 when `x` is less than `len`.

By inspection alone, it should be obvious that memory is the bottleneck because this loop does nothing else that is time-consuming. But, how bad is the bottleneck? Considering only the data dependencies, one time through the six steps of this loop can be executed in three clocks. But, timing this loop using 256,000 elements shows that it takes about nine clocks per element.

A performance experiment would be to remove the possibility of read cache misses to see what happens. The performance experiment code would look like:

```c
for (x=0; x<len; x++)
    DestArray[x] = SourceArray[0] + K;
```

Timing this new loop shows that the execution time drops to about five clocks per element. A second experiment would be to remove the possibility of write cache misses, as shown in the following sample.

```c
for (x=0; x<len; x++)
    DestArray[0] = SourceArray[x] + K;
```

The performance is now roughly four clocks per element. The final experiment with no cache misses follows.
for (x=0; x<len; x++)
DestArray[0] = SourceArray[0] + K;

This code executes in about two clocks per element. Compared with the original timing of nine clocks per element, accessing memory is causing the loop to be more than four times slower than the performance experiment with no memory accesses. This information leads to two conclusions. First, seven clocks per element are wasted waiting for cache misses. And secondly, write cache misses are a little more costly than read cache misses in this case. With this detailed analysis complete, it is time to start fixing the memory problems.

**Conclusion**

A memory problem occurs when the processor has to wait for its data. Typical causes of memory problems include page misses, store forwarding problems, and L1 cache misses. Page misses are caused when the processor is waiting for memory to be brought into the RAM for use by the application. Store forwarding problems are caused by disambiguation between stores and load addresses or by mismatches between writing memory size and reading memory size. L1 cache misses occur when the L1 cache is missing or is being accessed by multiple portions of the application. A performance monitor such as the VTune Performance Analyzer for Pentium 4 processors can be used to perform time-based sampling and identify hotspots for page misses, store forwarding problems, and L1 cache misses. In addition to this sampling analysis, performance experiments using sample code can be used to isolate memory, read cache misses, and write cache misses in order to determine the key performance bottlenecks. While the severity of these lapses varies and it is not always worthwhile to resolve all instances of the issue, this evaluation process is the first step in optimizing processor efficiency. For an excellent primer on multi-core processing and software optimization, consult the book from which this article was sampled.

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:


Also see our Recommended Reading List for similar topics:


**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.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to:

Copyright Clearance Center
222 Rosewood Drive, Danvers, MA 01923
978-750-8400, fax 978-750-4744

Portions of this work are from The Software Optimization Cookbook, Second Edition, by Richard Gerber, Aart J.C. Bik, Kevin B. Smith, and Xinmin Tian, published by Intel Press, Copyright 2011 Intel. All rights reserved.