White Paper

NewCode Technologies, Inc.

Memory Tuning System

MEMORY TUNING SYSTEM
NEWCODE TECHNOLOGIES, INC.

Copyright © 2001,2005 NewCode Technologies, Inc.

All rights reserved. NewCode and Memory Tuning System are registered trademarks of NewCode Technologies, Inc. All other trademarks are the property of their respective owners.

PUBLICATION DATE: JUNE 30, 2005

Address:	73 Concord Ave, Somerville, MA 02143 USA
Product Information:	(617) 513-8186
Fax:	(617) 591-9684
Web:	`http://www.newcodeinc.com`
E-Mail:	`sales@newcodeinc.com`

Contents

1 Introduction **2**

1.1 Tackling Allocator Inefficiencies 2

1.1.1 Options for Improving Application Throughput 3

1.1.2 Memory Allocation Tuning 4

2 Options for Increasing Application Performance **8**

2.1 Reduce Application Heap Usage 8

2.2 Scale Out, Not Up 8

2.3 Use NewCode MTS 9

3 Improving Memory Management and Performance **10**

4 Future of Scalable Applications **11**

5 Summary **12**

6 References **13**

Abstract

Over the past few years, significant changes have occurred in software and hardware architectures in regards to theoretical performance limits far exceeding what can be realistically achieved by todays applications. This paper will examine dynamic memory management, one of the most common performance barriers in modern software, and introduce a product, NewCode Technology Memory Tuning System (MTS).

NewCode MTS is a drop-in memory allocator that can dramatically improve the performance of applications in which memory is dynamically allocated and de-allocated. NewCode MTS improves application responsiveness and throughput under heavy loads by creating multiple virtual heaps that work with scheduled threads to reduce heap contention while minimizing lock contention. NewCode MTS is described as drop-in, as it requires no rework from the applications viewpoint – it simply replaces the existing memory allocation routines, requiring only a re-linking of the application with the NewCode MTS libraries rather than the system memory management library.

This paper will show how NewCode MTS offers applications significant performance gains through faster, scalable memory management and better hardware utilization.

Chapter 1

Introduction

The last decade has seen the introduction of sophisticated hardware architectures that promise significant performance increases. Unfortunately many of the hardware improvements, ranging from superscalar-pipelined CPUs to Symmetric Multiprocessing (SMP), require changes to software applications in order to fully realize the investment in such improvements.

On the software side, nearly all applications written today incorporate heavily object-oriented and multithreaded design patterns. These popular design idioms are important for delivering projects on time, to spec and within budget, but they arent without their problems. Object-oriented applications are notorious for their usage of the heap for dynamic memory allocation and multithreaded applications require careful attention to synchronization issues to protect the application state.

While memory requirements for applications vary greatly, well-written C/C++ applications have shown to spend anywhere from 5-40% of their time interacting with the heap, depending on the applications memory requirements and the efficiency of the allocator used[1].

1.1 Tackling Allocator Inefficiencies

Allocator efficiency issues are well known to software developers, since often it is the first item to tackle when performance tuning an application. Many techniques exist for dealing with some of the efficiency problems, such as allocating memory and never deallocating it.

While this technique works well for applications that are short-lived and work within known memory constraints during their lifetime, it would be rare to find a long-running application that could use this technique. Therefore, long-running applications need to find better methods of working around allocator inefficiencies. Another common method of working around allocator inefficiencies in C++ is to use what is known as a *Small Object Allocator* or *Memory Pool*. This method reserves large blocks of memory within the heap and uniquely manages these blocks for specific needs within the application. This can be significantly faster than using the standard heap and can dramatically improve overall application performance[2].

While the techniques listed above improve efficiency by reducing the interactions with the heap, they require special thought to be given to memory management from the start. In addition, they do not address an infamous problem that significantly impedes application scalability; default allocators found in most runtime libraries were designed before SMP machines became prominent, causing applications to suffer from poor scalability when operating in an SMP environment. This is in large part due to a global lock used to protect the heaps internal accounting structures from multiple threads using it simultaneously[3].

An application is only as scalable as its dependencies; therefore its easy to ascertain that an application using a non-scalable memory allocator will have scalability problems. When an application doesnt scale well, it wont be able to take advantage of SMP architectures. Doubling the number of processors will not double the performance of the application as one might expect.

Lets take a look at how applications using these default allocators are penalized in terms of scalability.

1.1.1 Options for Improving Application Throughput

A single-threaded on-line billing application running on a single processor machine can process 10 orders per second. A profiling tool shows that the application spends 15% of its time in `malloc()` and `free()` (the two most common heap operations.) It has been determined that the application must improve throughput to 40 orders per second to keep up with future growth in sales. What are the options for improving the throughput of this application?

Options:

1. Buy a faster machine. Since throughput in the on-line billing application is the number of orders per second, the system could be upgraded so that all of the components were four times faster. The CPU, hard disks and network interface would all need to be upgraded, as well as other dependencies such as databases and Webservers. For pedagogical purposes, add that the applications performance is not bound by anything other than the CPU. Therefore, to solve the problem the CPU would have to be upgraded to something four times faster. But, what if the CPU is the fastest one available in the market? According to Moores Law, a processor four times faster shouldnt be expected for another three years. Since waiting for a faster processor isnt usually an acceptable option, another solution needs to be found.

2. Buy a bigger machine. If the throughput cant be improved by buying a faster processor, the only available option is to increase the number of processing units. This is the SMP option: make the applications multithreaded so that four threads can perform the same order-processing tasks and then deploy on a four-way SMP machine. This would solve the throughput problems if each processor in the SMP machine could process 10 orders per second. However, upon close analysis this solution wont increase the throughput by the required amount. Why? This will be explained in the following section.

1.1.2 Memory Allocation Tuning

The online order-processing applications performance is not improved to forty orders per second by upgrading to a four-way SMP machine. This is due to the usage of the heap simultaneously from each of the four processors. Since only one of the processors can use the heap at any given moment, the performance of the application will be reduced by the amount of time spent waiting for access to the heap.

While 15% heap usage is a conservative average for C/C++ applications, the next section examines how it dramatically impacts the applications ability to be sped up.

Application Speedup

Back in the 1960s, Gene Amdahl discovered Amdahls Law, which correlates application performance to parallel computing. Roughly stated, Amdahl proved an applications performance was limited by the fraction of application code that must be run sequentially.

Since operations using primitive allocators must run sequentially due to the global lock discussed earlier, Amdahls law provides a way of quantifying how much performance is limited by the usage of the heap.

To determine the theoretical maximum speedup of an application, one can use Amdahls law as expressed by the following equation (c is the percentage of time spent in the allocator and P is the number of processors)[4]:

$$Speedup(c, P) = \frac{1}{c + \frac{(1-c)}{P}} \tag{1.1}$$

It is important to emphasize this is the theoretical maximum the application can be sped up, as therell undoubtedly be other sources of contention within the application besides the allocator that affect the scalability in the same way.

Application speedup is what development teams are after – an initial speedup of one should, in this example, lead to a target speedup of four. Therefore, for this application the maximum performance improvement with a four-way SMP machine would be:

$$Speedup(0.15, 4) = \frac{1}{.15 + \frac{.85}{4}} \approx 2.75 \tag{1.2}$$

By upgrading to four processors the application falls short of the target by a speedup of 1.25, or 12.5 orders per second. In fact, to reach the speedup goal of four, it would require nine processors.

This problem of diminishing return on investment can have a significant impact on the success of a business. In this online billing application, the system running the order-processing application became roughly nine times more costly to process just four times the current orders. The poor scalability of the application has made it difficult to predict the applications Total Cost of Ownership (TCO). In fact, if the amount of contention is unknown within the application it is impossible to predict the applications

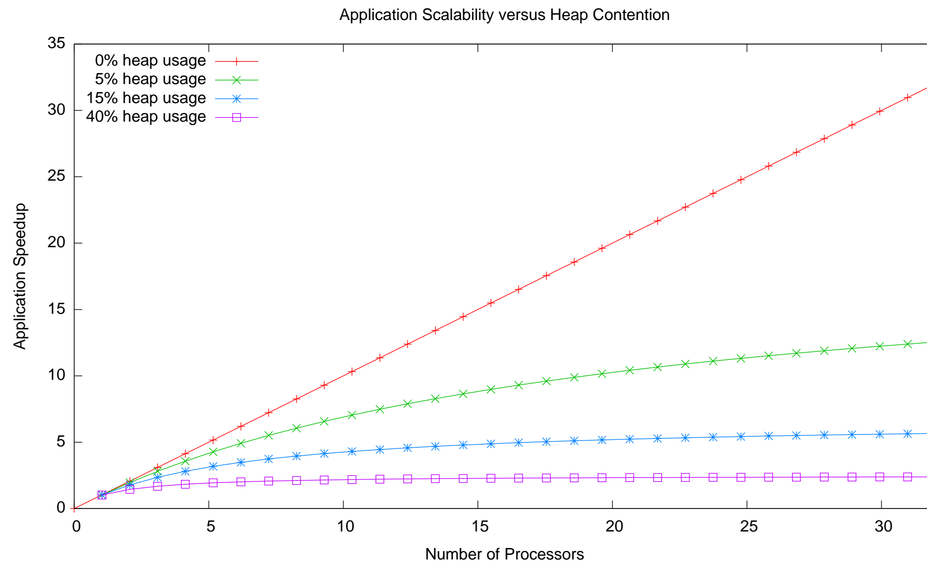

Figure 1.1: Application Scalability versus Heap Contention

TCO and an upgrade may negatively impact profitability.

Once plotted, the online billing application Speedup curve would look like the curve matching the 15% heap usage in Figure 1.1. In Figure 1.1, notice that the scalability is directly related to the amount of heap usage in the application.

In Figure 1.1, the scalability ”curve” for an application with perfect scalability is the line corresponding to an application with 0% heap usage. The performance within an application with linear scalability can be changed a predictable amount through a linear change in hardware resources. In this instance, to double the throughput of the machine, just double the size of the machine. However, performance improvement in typical C/C++ applications, where the heap usage is between 5-40%, isnt so easy with the addition of new hardware. From Figure 1.1, we can see that with as little as 5% heap usage, SMP machines above eight processors dont offer significant benefits for a single application.

NewCode MTS replaces the standard allocators with a proprietary, scalable allocator which most closely matches the theoretical performance expectations of SMP architectures and offers significant performance gains for single processor systems. Using NewCode MTS in the online billing application example would result in near linear scalability and predictable

TCO. In short, by using MTS a machine with only half the performance rating could be used with success.

It is important to note that without using NewCode MTS or reducing the usage of the heap to 0%, there will be an upper bound on the applications performance. This can be seen when looking at the online billing applications scalability curve from Figure 1.1. The scalability starts near linear, but then rapidly levels off to a maximum speedup of just over five. In fact, by taking the limit of $Speedup(c, P)$, as the number of processors approaches infinity, the precise upper bound for application scalability can be found.

$$\lim_{P \longrightarrow \infty} Speedup(c, P) = \frac{1}{c} \tag{1.3}$$

Therefore the upper limit of scalability for the online billing application without MTS is just 667% – the practical limit is quite a bit less since getting to six times speedup takes over 50 processors and the scalability of the OS and machine architecture will come into question.

What if the order processing application needed to process ten times the number of orders? Since this is above the application's upper limit of scalability, the problem remains of how to increase the performance gain from 425% to the full 1,000% (10x gain).

Chapter 2

Options for Increasing Application Performance

This chapter addresses the options for eliminating scalability and performance problems due to dynamic memory allocation.

2.1 Reduce Application Heap Usage

Code tuning can be used to reduce the applications heap usage from 15% down to nearly 5%. However, even with a reduction from 15% to 5%, twenty processors would be necessary to achieve the throughput our order processing application requires.

2.2 Scale Out, Not Up

Instead of purchasing a large 10-processor SMP box, one could buy ten or more single processor boxes. This might be a good solution if the growth in order processing isnt needed immediately, therefore allowing time for redesigning the application to run on a group of machines.

Some refer to this optimization as ”Scaling Out” instead of ”Scaling Up” as with SMP machines. This solution isnt always practical; besides the significant cost and risks associated with rewriting the application at the most costly time in its lifecycle[5], increased system administration costs will be

incurred and additional software or hardware support will be required for fault-tolerance and load-balancing.

2.3 Use NewCode MTS

Plug NewCode MTS into the online order processing application to reduce the synchronized heap access to 0%. This reduction would remove the thread heap contention within the application and provide near-linear scalability. Using NewCode MTS, the throughput of the order processing application is improved 1,000% with as few as ten processors.

Additionally, NewCode MTS doesn't require costly modifications to existing applications, as it doesnt require modifying the application late in the product life cycle. NewCode MTS is simply linked into the application, with no changes to application code required.

To more fully quantify the performance NewCode MTS can provide, the following equation can be used for approximating the benefit of using MTS:

$$MTS(c, P) = \frac{P}{Speedup(c, P)} \tag{2.1}$$

where c is the percentage of time currently spent using `malloc()` and `free()` (or, the operators `new` and `delete` in C++) and P is the number of processors.

In the online billing application, 425% speedup was achieved on a 10-processor SMP box when the application used the standard memory allocator routines. By simply relinking the application with NewCode MTS, an additional benefit of 235% – above the 425% already gained — could be achieved, for an overall increase in throughput of 1000%.

Chapter 3

Improving Memory Management and Performance

Todays compiler runtime libraries arent efficient on SMP systems, and are a critical bottleneck that limits a multithreaded application from fully utilizing the SMP architecture and running at potential speed. As described previously, NewCode MTS addresses this deficiency, allowing applications executing on SMP architectures to not be constrained by memory allocation and helping to maximize the usage of all processors in the system.

The value of NewCode MTS isnt limited to applications running on SMP architectures. Since MTS consumes as little as 25% of the machine cycles per memory allocation compared to most compiler runtime allocators, it can offer significant value to applications running on single processor machines. Long running, multithreaded applications will also reap the benefits of improved spatial and temporal memory reference locality, which has been shown to increase application performance by an additional 25% [6].

Memory management is only one component of a systems dynamic performance; however, it can represent one of the most serious application performance bottlenecks. Development teams building high-performance, mission-critical applications find themselves spending valuable resources on additional hardware and memory management customization.

Since the allocators usage can be easily determined from profiling utilities and is frequently used in abundance within C/C++ applications, it remains the focus of this document. There are other areas that may be tuned in applications, but none are as consumptive of performance or as easy to tune as the memory allocator.

Chapter 4

Future of Scalable Applications

While SMP systems are becoming more popular, there are still application developers that ignore scalability concerns when targeting every-day desktop computers. There is a problem for these developers on the horizon, because of recent trends in microprocessor design. The problem stems from the fact that as processors get many times faster than the components they are connected to, processors will be spending more of their time waiting and less time processing. Processor manufactures have begun incorporating technologies within their CPUs to take advantage of this fact.

Intels Hyper-threading is an example of a technology taking advantage of extra CPU bandwidth. This technology allows one CPU to emulate many logical processors. The advantage of Hyper-threading is that it offers up to a 65% performance improvement [7] for about the same cost as a processor without Hyper-threading. The disadvantage is that many applications that take advantage of dynamic memory allocation will not offer their users the performance they expect out of these processors.

As more processor manufacturers incorporate multicore technologies like Hyper-threading itll become more important to consider scalability issues in standard desktop applications.

Chapter 5

Summary

This paper examined how the heap usage found in multithreaded, object-oriented software applications can have dramatic effects on runtime performance and scalability. Additionally, a correlation between application scalability and total cost of ownership was drawn.

To solve these scalability issues with the heap, the multiple options available were examined: upgrading hardware or increasing the number of processors on an SMP system. Each of these options has significant shortcomings, including prohibitive costs, application scalability and finite limits of performance.

In short, a method of improving performance without costly hardware changes requires a significant application reengineering effort that could carry enormous project risk if time-to-market is essential. Solutions that minimize engineering efforts and hardware costs are valuable.

As a more efficient answer, NewCode MTS offers a drop-in solution that requires virtually no changes to an application, minimizing the risk of delayed deployment due to application performance. MTS simplifies application performance tuning, whether it be instantly solving application performance problems or diverting strategic resources from costly hardware purchases to real business issues.

Visit http://www.newcodeinc.com/ for information on NewCode MTS and to try a free evaluation.

Chapter 6

References

[1] David Detlefs; Al Dosser; Benjamin Zorn, *Memory Allocation Costs in Large C and C++ Programs*, 1993

[2] Andrei Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison Wesley Professional, 2001

[3] Per-ke Larson; Murali Krishnan, *Memory Allocation for Long-Running Server Applications*, ACM ISMM98, pp. 176-85, 1998

[4] John L. Hennessy; David A. Patterson, *Computer Organization and Design*, Morgan Kaufmann Publishers, 1998

[5] Steve McConnell, *Code Complete*, Microsoft Press, p. 26, 1993

[6] Dirk Grunwald; Benjamin Zorn; Robert Henderson, *Improving the Cache Locality of Memory Allocation*, ACM PLDI93, pp. 177-86, 1993

[7] Marr, D.; Binns, F.; Hill, D.; Hinton, G.; Koufaty, D.; Miller, J.; Upton, M., *Hyper-Threading Technology Architecture and Microarchitecture: A Hypertext History*, Intel Technology Journal http://developer.intel.com/technology/itj/2002/volume06issue01/, 2002