XMALLOC: A SCALABLE LOCK-FREE DYNAMIC MEMORY ALLOCATOR FOR MANY-CORE MACHINES

BY

XIAOHUANG HUANG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Advisor:

Professor Wen-mei Hwu

Abstract

There are two venues for many-core machines to gain higher performance: increasing the number of processors and number of vector units in one SIMD processor. **A truly scalable algorithm should take advantage for both venues**. However, most of past research, on scalable memory allocators such as atomic operation based lock-free algorithms, can be scalable with number of processors growing, but have poor scalability with the number of vector units in one SIMD processor growing. As a result, they are not truly scalable in many-core architecture.

In this work, we introduce our proposed solution used in the design of XMalloc, an truly scalable, efficient lock-free memory allocator. We will present (1) Our solution for transforming traditional atomic CAS(Compare-And-Swap) based lock-free algorithm to be truly scalable for many-core architecture. (2) A hierarchical cache-like buffer solution to reduce the average latency for accessing non-scalable or slow resource such as the memory system in many-core machine.

We used XMalloc as a memory allocator for NVIDIA Tesla C1600 with 240 processing units. Our experimental results show that XMalloc achieves very good scalability in terms of the number of processors and the number of vector units in each SIMD processor growing. Our truly scalability lock-free solution achieve 211 times speedup comparing to the common lock-free solution.

To Father and Mother.

Acknowledgments

This project would not have been possible without the support of many people. Many thanks to my adviser, Wen-mei Hwu, who read my numerous revisions and helped make some sense of the confusion. Also thanks to my colleagues, Christopher I. Rodrigues and Sara S. Baghsorkhi, who a lot of valuable suggestion. Thanks to the University of Illinois Graduate College for providing me with the opportunity to study and complete my degree here. And finally, thanks to my parents, and numerous friends who endured this long process with me, always offering support and love.

Table of Contents

List of Figures

List of Abbreviations

GPU	Graphics Processing Unit.
CPU	Central Processing Unit.
SIMD	Single Instruction, Multiple Data.
CAS	Compare And Swap.
NVIDIA	is a multinational corporation which specializes in the development of graphics processing units and chipset technologies for workstations, personal computers, and mobile devices.
CUDA	an abbreviation for Compute Unified Device Architecture, a parallel programming framework by NVIDIA.
OpenCL	is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors.

List of Symbols

τ	Time taken to drink one cup of coffee.
μg	Micrograms (of caffeine, generally).

Chapter 1

Introduction & Background

1.1 Introduction

Dynamic memory allocators, are widely used in sequential and multi-threaded applications. However many-core programming models, such as CUDA, OpenCL, still cannot provide programmers any dynamic memory allocator in parallel.It becomes a barrier for designing a C++ like object orientated massive parallel programming language. For shared-memory programs with hundreds or thousands of concurrent threads, the lack of support for dynamic memory management often emerges as a programmability obstacle. Without a safe and scalable memory allocator, programmers are forced to estimate the dynamic memory usage and pre-allocate the memory space in advance, which can be either error prone, inefficient, or both.

The XMalloc allocator algorithm scale with the number of processors and the number of vector units in a processor. We design a atomic CAS based lock-free algorithm to get excellent scalability with number of processors. Then we provide a solution to transform traditional CAS based lock-free algorithm to be scalable SIMD processing units. We coalesce atomic operations to achieve better data parallel execution. In this way, our algorithm is also scalable with number of vector units in each processor. Our solution achieves superlinear speedup.

1.2 Background and Related Work

On shared-memory multi-core or many-core machines, many processing units share the same data structure. To ensure the consistency of these concurrent objects, processes need a mechanism for synchronizing their access. In such a machine the programmers typically have to use synchronization primitives such as semaphores, monitors, guarded statements, mutex locks, but algorithms based on locks do not scale well. Consequently the operations of different processes on a shared data structure should appear to be serialized: if two operations execute simultaneously, the system guarantees the same result as if one of them is arbitrarily executed before the other. [**?**]

Atomic Instructions perform read-modify-write atomic operations on one 32-bit or 64-bit word residing in memory. The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads. In other words, no other thread can access this address until the operation is complete. The widely used lock-free solutions [9] [8] [10] are based on atomic operation based.

The research about lock-free synchronization started from more than two decades before. For example, figure 2.2(a) is an example of novel CAS based lock-free algorithm. It is attributed to early work by Lamport [5]. The impossibility and universality results of Herlihy [4] had significant influence on the theory and practice of lock-free synchronization, by showing that atomic instructions such as CAS and LL/SC are more powerful than others such as Test-and-Set, Swap, and Fetch-and-Add, in their ability to provide lock-free implementations of arbitrary object types Michael and Scott. [7] reviews practical lock-free algorithms for dynamic data structures in light of recent advances in lock-free memory management.

A lot of memory allocation research in sequential or multi-threaded environment has been done before. Ptmalloc [3], developed by Wolfram Gloger and based on Doug Leas dlmalloc sequential allocator [6], Hoard [1], developed by Emery Berger are lock-based memory allocator. Mostly Lock-free malloc, designed by Dice and Garthwaite [2] propose a partly lock-free allocator.

Hoard [1], uses multiple processor heaps in addition to a global heap. Each heap contains zero or more superblocks. Each superblock contains one or more blocks of the same size. Statistics are maintained individually for each superblock as well as collectively for the superblocks of each heap. When a processor heap is found to have too much available space, one of its superblocks is moved to the global heap. When a thread finds that its processor heap does not have available blocks of the desired size, it checks if any superblocks of the desired size are available in the global heap. Threads use their thread ids to decide which processor heap to use for malloc. For free, a thread must return the block to its original superblock and update the fullness statistics for the superblock as well as the heap that owns it. Typically, malloc and free require one and two lock acquisitions, respectively.

Michael [8] presents a completely lock-free memory allocator which guarantees progress regardless of whether some threads are delayed or even killed and it is immune to deadlock. It uses atomic CAS(Compare-And-Swap) instructions to implement a lock-free algorithm to allocate memory resource. It uses some high-level structures from Hoard, and merge the data structure into 64-bit data which can use one atomic operation to update. It use heaps to maintain the superblocks, and allocate a fixed size block from superblock.

Chapter 2

Detail of XMalloc

2.1 Overview of XMalloc Design

The XMalloc algorithm manages a hierarchical memory pool. The memory pool is divided into a number of *Memoryblocks*. Each *Memoryblock* can be divided into *Memoryblocks* and multiple *Memoryblocks* can also be merged to be one *Memoryblock*. *Memoryblocks* are distributed on different heaps according to their sizes. Each *Memoryblock* container multiple *Superblocks*. Each *Superblock* contains 1-32 equal-sized blocks, we call them *Basicblocks*.

In order to reduce the average memory allocation time for each thread, XMalloc maintains a two-level lock-free FIFO buffer. The first level buffer, which contains *Basicblocks*, is fast and scalable but the size of memory that can be served from this level is fixed at a small size. Memory allocation requests that are small in size are served out of the first level buffer. The rationale is that the vast majority of memory allocation requests by user programs are small in size. Therefore, most memory allocation requests can be serviced by the first level buffer to achieve high speed and scalability.

The second level buffer, which is used to contain *Superblocks*, is slower but the memory allocation size that can be served by this level is variable and can be large. Large memory allocation requests are served out of the second level buffer. Furthermore, when the first level buffer runs out of *Basicblocks*, XMalloc removes *Superblocks* from the second level buffer, breaks them down into *Basicblocks*, and fill the *Basicblocks* into the first level buffer. The access to the second level buffer is much slower and less scalable than those to the first level. As long as the portion of the memory allocations served by the second level buffer is kept small, the average speed of memory allocation will be high.

2.1.1 Achieving SIMD Scalability

Most many-core GPU architectures use SIMD processors to achieve higher computing throughput. SIMD designs reduce hardware cost by exploiting data parallelism. However, when a program has branch divergence, the execution efficiency of SIMD processors diminishes quickly. Furthermore, operations that access shared resources, such as

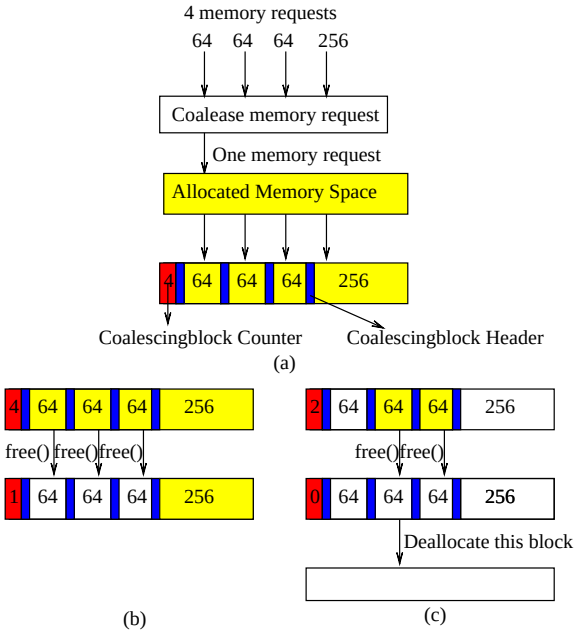

Figure 2.1: In the figure, yellow indicates that a block is allocated; white indicates that a block is empty. (a) Four threads request for memory spaces of 64 bytes, 64 bytes, 64 bytes and 256 bytes simultaneously. XMalloc coalesces them to be one memory request. Once a memory block is returned, all threads will pick up the portions for their allocation requests from that memory block. simultaneously without branch divergence. (b) During memory de-allocation, if only parts of a block is de-allocated, we only decrease the value of $counter$. (c) If all parts of a block is deallocated, we will deallocate the whole block and return it to the FIFO buffer.

```
do { // Repeat until CAS succeeds
     oldvalue = *addr;
     // Critical section
     newvalue = f(oldvalue);
}while(CAS(addr, oldvalue, newvalue));
     (a)

do { // Repeat until CAS succeeds
     oldvalue = *addr;
     // Critical section
     newvalue = SIMD_reduce(f, oldvalue);
     if (ThreadID == 0)
          *success = CAS(addr, oldvalue, newvalue);
} while (!(*success));
     (b)
```

Figure 2.2: CAS-based lock-free algorithm: (a)Traditional CAS-based lock-free algorithm code. This simple CAS-based lock-free algorithm is not scalable in the SIMD dimension. (b) The SIMD scalable version. The algorithm coalesces multiple allocation requests into a single one to eliminate branch divergence due to the CAS operation.

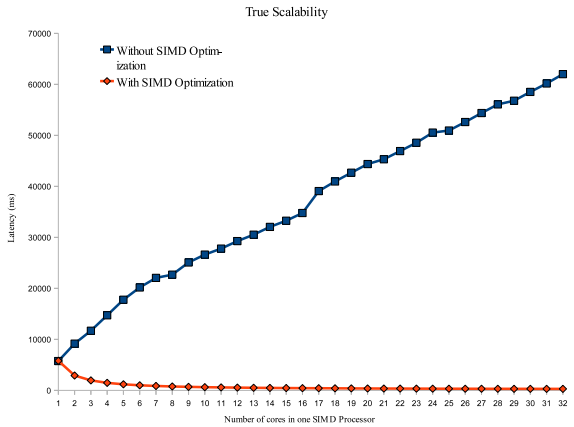

Figure 2.3: Comparison between true scalable lock-free algorithm and traditional scalable lock-free algorithm.

atomic operations, on SIMD processor will become much slower if all of the concurrent threads in the same SIMD processor try to obtain the same resource.

2.1.2 The Problem of Atomic CAS Based Lock-free Algorithm

Many scalable parallel algorithms follow the pattern shown in Figure 2.2(a), where a critical section begins by reading a shared variable and ends by updating the variable with a CAS operation. If the update CAS fails, meaning that the variable was modified by another thread, then the section is repeated. When several threads execute the critical section simultaneously, one thread will finish first and update the variable. The lagging threads' CAS operations will all fail, and these threads will repeat the critical section.

SIMD implementation of thread execution produces pathological performance for such code. Because all SIMD threads on one processor execute in lockstep, they execute the critical section simultaneously; one thread will complete the critical section while the others fail. An N-wide SIMD processor will loop through the critical section at least N times, as slow as if all SIMD threads ran serially. What is worse, because multiple SIMD threads are active in each iteration, the loop body is executed at least $\frac{(N+1)*N}{2}$ times, generating more memory traffic than serial execution would. When memory bandwidth is a bottleneck, this will slow down the execution. The more vector units each SIMD processor has, the slower execution becomes.

The blue curve in Figure 2.3 demonstrates the issue, using our atomic CAS based lock-free allocator as an example. In this benchmark, we generate 1 million threads, each of which makes one call to malloc and one call to free. We

simulate varying SIMD width by running the computation on a subset of threads in each warp. Using a simple CAS based lock-free algorithm, the execution latency rises rapidly with SIMD width. The performance is about 11 times slower if we use SIMD processors with 32 vector units compared to a serial processor. Whereas the execution latency continue to decrease when we use our proposed scalable algorithm, as shown in the orange curve in Figure 2.3.

2.1.3 Scalability Transformation for SIMD Processors

The cause of poor SIMD scalability of the simple CAS code is that all SIMD threads in a processor execute a critical section simultaneously, thus inducing worst-case contention for synchronization variables. For critical sections that consist of reads followed by a single CAS operation to a synchronization variable (Figure 2.2(a)), the problem can be solved by combining the transactions from all SIMD threads into one. Since all transactions update the same variable, the combined transaction also updates a single variable and can be committed by issuing a single CAS from one SIMD thread (Figure 2.2(b)).

Combining the transactions involves making a local copy of the synchronization variable, accumulating updates from all SIMD threads, and then committing the result globally. Commutative and associative updates can be accumulated using SIMD reduction instructions or a reduction tree. When updates cannot be expressed as a reduction, one thread can sequentially update the local copies of the synchronization variable. Loads and thread-local computation may remain parallel. Executing a single CAS instruction for all SIMD threads in a warp relieves the problem of branch divergence and memory contention. This leads to the following key observation.

In an SIMD scalable lock-free algorithm, no two threads in the same SIMD processor should apply atomic CAS on the same address, if that atomic CAS operation is the condition for a loop.

The loop body of our CAS-based lock-free algorithm is much more complex than the example we show in figure 2.2. There are several of CAS based atomic loops and nested CAS based atomic loops. If we try to transform them one by one, the procedure can be very complex and error prone.

Therefore, we choose to merge the memory allocation requests from all the threads in a SIMD processor into one memory allocation request. Then we have only one thread to call the allocation algorithm. After that, each thread will pick up its own piece with its thread ID from the returned memory block in parallel without branch divergence. In Figure 2.1(a), the blue line is the performance of XMalloc without SIMD scalable transformation, the orange line is the performance of our truly scalable version. The SIMD scalable version runs up to 211 times faster.

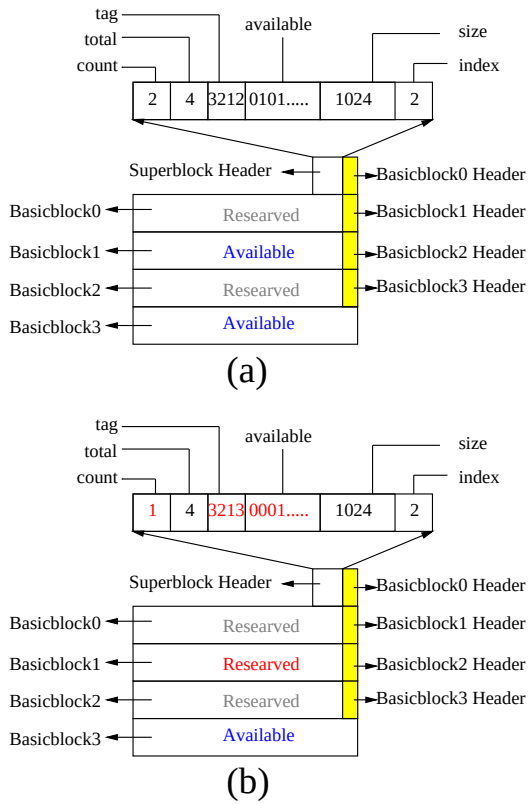

Figure 2.4: Superblock Structure: (a) This is a $Superblock$ contains 4 $Basicblocks$. $Basicblock1$ and $Basicblock3$ still available, so the first 4 bits of $available$ is 0101. (b) This the $Superblock$ after we reserve the $Basicblock1$ from (a), the red parts are the data which are changed. We use one 64-bit atomic CAS to update them.

2.2 Two Level Buffered XMalloc Algorithm

The number of cores in many-core machine is growing much faster than the performance of the memory system. If a operation accesses the slow memory system, it can cause major performance degradation when executed by massive number of threads in a many-core machine.

Memory allocation is used to manage shared resource, if thousands of memory allocation requests are made simultaneously, the shared resource will become the bottleneck of performance. In order to reduce the average memory request latency in many-core machine, we introduce a hieratical cache-like solution to speed up our XMalloc allocator. We use two level buffers to reduce the impact on operations with bad scalability and achieve better parallelism and performance. The first level buffer is faster but can only satisfy one fixed size memory allocation request by one operation. The second level buffer is slower but can generate up to 32 memory space to put into first level buffer by one operation, so the latency is also very small if we divide it by 32.

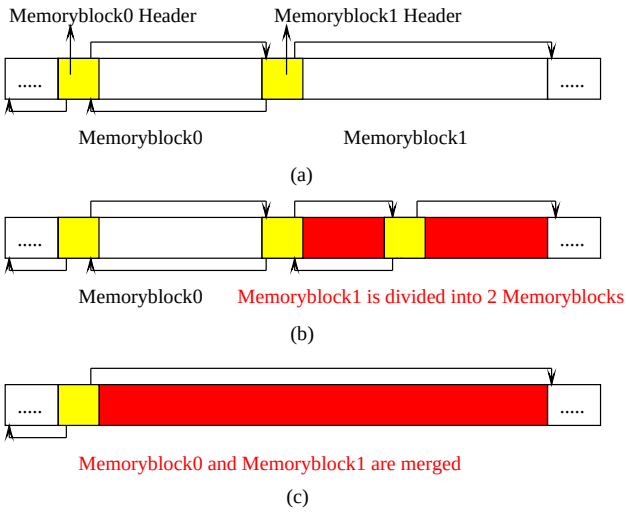

Figure 2.5: Memoryblock Structure: (a) shows 2 continuous Memoryblocks. Each Memoryblock has pointers to point to its predecessor and successor, which are also neighbors. (b) shows Memoryblock1 is divided into 2 Memoryblocks. (c) show Memoryblock0 and Memoryblock1 are merged into one Memoryblock.

2.2.1 Data Structure

There are four kinds of memory blocks:

- **Basicblock** is a fixed size memory space. It is used for small memory allocation request, also can be used as container of $Coalescingblocks$.

- **Superblock** is container for $Basicblocks$. We show an example of $Superblock$ in figure 2.4(a).

- **Memoryblock** is the data structure to manage the memory pool resource, it can be in any size. All the $Memoryblocks$ are linked together with bidirectional link. A $Memoryblock$'s predecessor and successor are its neighbors. It is used for large memory space request, also can be container of a $Superblock$ or $Coalescingblocks$. See an example in figure 2.5(a).

- **Coalescingblock** is a special data structure for our SIMD scalability optimization. $Coalescingblocks$ is used as a memory block for any kind of memory request. One $Memoryblock$ or $Basicblock$ can contain serval $Coalescingblocks$. See the example on figure 2.1(a).

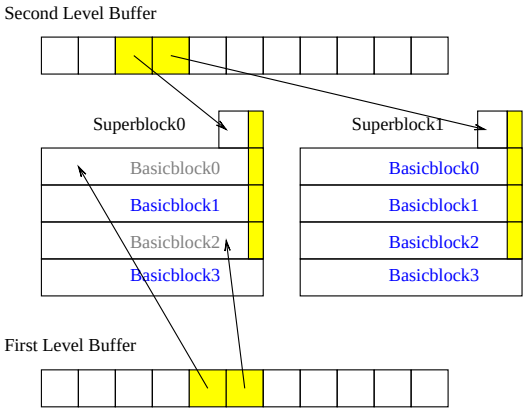

Figure 2.6: Example of two level buffer

2.2.2 Algorithm

In many-core programming models such as CUDA and OpenCL, thousands of memory requests will be made simultaneity. Even the previous best scalable dynamic memory allocation algorithm, the latency multiply by thousands, will become intolerable. In order to get higher throughput to match the many-core architectures, we designed our two-level buffer to achieve high throughput.

- **Allocation Algorithm:**

  The main idea for our algorithm is that, when there is a memory request, we will remove a corresponding $Basicblock$ from first level buffer, which is very fast and scalable. If first level buffer run out of resource, we will remove a $Superblock$ from second level buffer, then we can get 1-32 $Basicblocks$ from the $Superblock$ to fill the first level buffer. If the second level buffer is still empty, we will allocate $Superblocks$ from $Memoryblock$ heap.

- **Deallocation Algorithm:**

  At first,we will read the memory flag, which will indicate the memory block type.

  If it is $Basicblock$, we just enqueue it to the first level buffer. If the first level buffer is full, we will return it to its $Superblock$ and enqueue it to the second level buffer.

  For Memoryblock, we just free it, then to check whether its predecessor or successor is also empty, if so, merge them. Do it repeatedly until neither its predecessor nor successor is empty.

Memoryblock Operation

Memoryblock is the original memory from global memory, it can be any size. All the physical blocks are linked together with bidirectional link, as figure 2.5(a). All the $Memoryblocks$ are put in a heap. If there is a memory request, we will pick one of the fit $Memoryblock$ from the heap. If the $Memoryblock$ is too big. We will divided it into two pieces and save back the rest part to the heap. When we return $Memoryblock$ to the heap, we will merge two $Memoryblock$ in the heap if they are neighbors. figure 2.5 illustrates these operations.

Chapter 3

Experimental Result and Summary

3.1 Experimental Result

Our experimental performance results are derived on a NVIDIA Tesla C1600, with CUDA 3.0 64-bit compiler in linux. We use a benchmark $MilThreads$, which creates 1 million threads, each thread will allocate 8 bytes of memory, then write 8 bytes to that space, then deallocate the space.

3.1.1 Latency

Here,we compare the allocation latency with NVIDIA API $cudaMalloc$ in CUDA 3.0. $cudaMalloc$ is provided by NVIDIA to allocate GPU memory in the CPU. It is sequential function. So our XMalloc allocates memory in the test cases with only one thread running on the GPU. The test result show that NVIDIA API $cudaMalloc$ has a 0.18 ms latency. Our algorithm has a latency of about 0.05 ms with only one thread. The size of memory request is from 1 byte to 32M.

For average allocation time in parallel for each thread. We test from 1 thread to 1024 concurrent threads that request for memory with 8 bytes. For 1 thread, it needs 0.04 ms to finish allocation. But for 32 threads, the average allocation time for per thread is 0.0013ms, and for 1024 threads case, only need 0.00015ms per thread.

Figure 3.1 shows that the speedup trend as the the number of processors increases. The blue line is XMalloc solution with the two-level buffer, the orange line is XMalloc solution without buffer.(Both of them use SIMD CAS coalescing). From the test data, we can see that, the ”with buffer” version’s scalability is better ”without buffer” version.

3.1.2 Scalability

We test the scalability according to Tesla C1600 which has 240 cores and each core can run 4 threads simultaneously. So we can have can have 960 simultaneous threads at most. We test XMalloc with different numbers of simultaneous

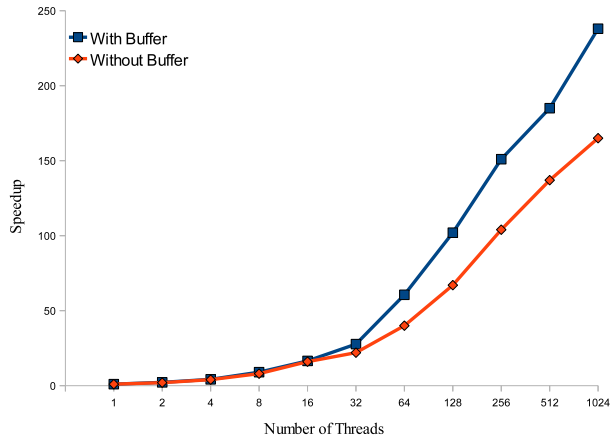

Figure 3.1: Average Allocation Latency per Thread

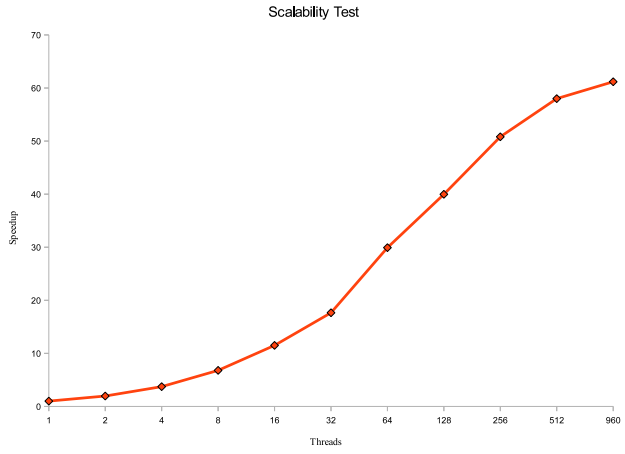

Figure 3.2: Scalability test. 1 million malloc/free pairs. The speedup is comparing to sequential latency.

threads, the result show that, the scalability is pretty good.
Besides, for test truly scalable feature, we test the scalability with number of processors growing and number of vector unit in one SIMD processor growing.

Figure 2.3 shows the latency with increasing number of vector units in each SIMD processor for benchmark $MilThreads$. Orange line is the XMalloc truly scalable solution, it gets 20 times speedup when vector units per processor increases to 32. The blue line is the naive XMalloc solution without SIMD scalability solution, It is 10 times slower than a serial processor when vector units per processor grows up to 32. It shows that our SIMD scalable solution is 210 times faster than the traditional scalable version.

3.1.3 Two Level Buffer Improvement

As figure 3.1 shows the speed growing with the number of processors. The blue line is XMalloc solution with buffer, the orange line is XMalloc solution without buffer. From the test data, we can see that, the "with buffer" version's scalability is better "without buffer" version.
In order to test the benefit of the two level buffer system. We have a naive version of memory allocator for $Memoryblocks$ with bad scalability, as the blue line in figure 3.2. Then we use the two level buffer to speed up the naive allocator, the performance is the orange line. From the performance graph, the scalability is improved a lot by the two level buffer a lot. For 960 concurrent threads, the two buffers version is 9 times faster than naive version.

3.1.4 Space Efficiency

Discussing about the internal fragmentation. For small memory space request the maximum internal fragmentation is 50%, because the size of Basicblock is growing by factor 2. (But there is worst case, if the program request for 1 byte, our minimum Basicblock is 60 bytes. Then we waste 59 bytes.) The Basicblock can be fully reused after deallocation. No external fragmentation issue on this, because the size of Basicblock is fixed and allocate from Superblock directly. If the programmers really want to reduce the internal fragmentation, they can tune the growing factor. For big space request, we will allocate Memoryblock for it. The only extra memory consuming of Memoryblock is the data structure. So the internal fragmentation is very small. To reduce the external fragmentation, our algorithm can merge neighbor Memoryblocks if both of them are empty.

3.2 Summary

In this paper we presented a completely lock-free dynamic memory allocator for many-threaded architecture, XMalloc. Being completely lock-free, our allocator is immune to deadlock free. Our four stages allocator can provide very high throughput. We only use atomic ADD and atomic CAS operations which are widely supported by multi-core and many-core machine. Besides, we use standard c grammar which is not depended on hardware or compiler. What is more, our solution can be applied on a system without any memory allocator. Our special solution for SIMD instructions call malloc, greatly speedup the performance and benefit the memory usage.

From the experimental result, our XMalloc algorithm is scale with cores and SIMD width. It is suitable for many-core architecture such as GPU.

References

[1] Emery Berger, Kathryn McKinley, Robert Blumofe, and Paul Wilson. Hoard: A scalable memory allocator for multithreaded applications. Technical report, Austin, TX, USA, 2000.

[2] Dave Dice and Alex Garthwaite. Mostly lock-free mostly. In *ISMM '02: Proceedings of the 3rd international symposium on Memory management*, pages 163–174, New York, NY, USA, 2002. ACM.

[3] Wolfram Gloger. Dynamic memory allocator implementations in linux system libraries. http://www.dent.med.uni-muenchen.de/ wmglo/.

[4] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[5] Leslie Lamport and Robert W. Taylor. Concurrent reading and writing of clocks. *ACM Transactions on Computer Systems*, 8:305–310, 1990.

[6] Doug Lea. A memory allocator. `http://gee.cs.oswego.edu/dl/html/malloc.html`.

[7] Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 21–30, New York, NY, USA, 2002. ACM.

[8] Maged M. Michael. Scalable lock-free dynamic memory allocation. *SIGPLAN Not.*, 39(6):35–46, 2004.

[9] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. 15th ACM Symp. on Principles of Distributed Computing*, pages 267–275, 1996.

[10] Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *SPAA '01: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 134–143, New York, NY, USA, 2001. ACM.

Author's Biography

Xiaohuang Huang was born in China. He got B.A degree in Zhejiang University, China in 2007 and studied in Hong Kong University of Science & Technology during 2005-2006 in Computer Science. His research area in Undergraduate is Computer Graphic. After he came to UIUC, he started to do Supercomputing and Compiler research based on GPU. He has ever worked in Microsoft Research for GPU compiler on Phoenix Framework and worked as CUDA Developer in NVIDIA.