Impact of Various Memory Allocation Techniques on Range Coding Performance

Darko Trivun, Alvin Huseinović, Merisa Huseinović, Arnes Durmo

***Abstract* — Shared memory multiprocessor systems need efficient memory allocators, primarily for parallel programming support. In first part of this paper, comparison between two memory allocation libraries is presented. Benchmark performing processing similar to backend of web application is used. Each benchmark was repeated twice, once with each library. Libraries used are *jemalloc* and *dlmalloc*. In second part of this paper, benchmarks were done to show if range coding algorithms benefit from different dynamic memory allocation techniques.**

***Keywords* — benchmarking, dlmalloc, dynamic memory allocation, jemalloc, malloc, range coding**

I. Introduction

PERFORMANCE of a memory allocator can be benchmarked by measuring the average or peak usage of memory in a certain time-frame. In order to make a benchmark results useful, it is not enough to measure the amount of time that memory is being allocated by an application. Furthermore, the organization of memory can affect application's execution time. In first part of this paper, allocators were compared by measuring amount of useful work done in certain amount of time. Test suite cannot cover all possible cases, so an allocator may behave worse than expected. Regardless, an allocator has to operate well for certain amount of work and because of that, it is very important that a benchmark uses wide variety of allocation patterns.

Fragmentation has some side-effects regarding performance, all depending on application's execution profile. There are two types of fragmentation: internal and external. Internal occurs when a process allocates more memory than it requires. The rest of the allocated memory remains unavailable to rest of the programs until the process makes it available, and that might happen only when that process terminates. External fragmentation is usually present in systems which are aiming to achieve the continuous memory allocation.

Because of that, processes are often allocating and freeing varying amounts of the memory, and over time, memory becomes divided in the small pieces. When a process cannot allocate whole amount of memory it needs due to the small size of free memory blocks, external fragmentation occurs. It happens even though there is enough free memory for the process. Both types of fragmentation have different impacts on performance, depending on nature of given application [1]. In best case, both types of fragmentation should not occur. A memory allocator can also copy chunks of data between the different locations in memory, and that can also influences performance and continuality of memory allocation.

Implementation of *jemalloc* takes into account that virtual memory paging can cause the performance degradation. Even simple fetching from the RAM adds considerable latency when compared to the cache access, which then causes further performance degradation. In case when the work set is not cached, general performance will be better if it is continually placed within the cache blocks. Objects that are allocated in close time intervals will very likely be used at the same time, so if an allocator can allocate memory for such objects continually, performance will be better in the long run [2]. Rather than minimising total memory consumption, first priority of *jemalloc* is achieving continual allocation.

One of the main problems of SMP systems is the resource handling. As the number of processors is growing, the competition for resources is growing too. When system contains fewer cores, there is less competition, so general performances are higher [3]. As the number of processors is increasing, competing between the processors causes the performance degradation, so we may say that scalability is eventually decreasing.

It is the same with the SMP systems where every processor has its own cache, where it comes to further serious problems which affect an application's performance. The basic requirement of those systems is that cache of every processor shares the same memory address space. That means that data one processor's cache can access has to be same as data other processor's cache can access and so on. To make this possible, memory is divided into smaller parts. Each location in the cache has a tag along with the index of the block address in main memory that has been cached. These entries are called cache lines or cache blocks and are located in the cache [4].

Cache line and its status can be controlled by the processor cache, so if one processor wants to use a cache line, it first has to make all copies of this line that exist in cache memories of the other processors invalid. By

Darko Trivun is with the Faculty of Electrical Engineering (Department of Control Theory and Electronics), University of Sarajevo, Bosnia and Herzegovina (e-mail: dtrivun@etf.unsa.ba)

Alvin Huseinović is with the Faculty of Electrical Engineering (Department of Computer Science and Informatics), University of Sarajevo, Bosnia and Herzegovina (e-mail: ahuseinovic@etf.unsa.ba)

Merisa Huseinović is with the Faculty of Electrical Engineering (Department of Computer Science and Informatics), University of Sarajevo, Bosnia and Herzegovina (e-mail: mhuseinovic@etf.unsa.ba)

Arnes Durmo is with the Faculty of Electrical Engineering (Department of Computer Science and Informatics), University of Sarajevo, Bosnia and Herzegovina (e-mail: adurmo@etf.unsa.ba)

Novica Nosović is mentoring professor of this paper and is with the Faculty of Electrical Engineering (Department of Computer Science and Informatics), University of Sarajevo, Bosnia and Herzegovina (e-mail: nnosovic@etf.unsa.ba)

doing this, the processor obtains appropriate rights, and only then, that block of data may be changed [4].

The allocation arena is a set of blocks, each of which consists of a data area (whose addresses are returned by the allocation functions) immediately preceded by a header which describes the block's size and status. In most implementations, a block's size is the distance in bytes from its header to the next block's header, and each size is normalised to be a multiple of a minimum internal alignment. From an application's viewpoint, this can easily result in blocks with spare bytes at the end of the data area. [5]

If two threads are being run by a separate processors and are trying to manipulate different objects which are in a same cache line, then they must fight over the ownership (figure 1).

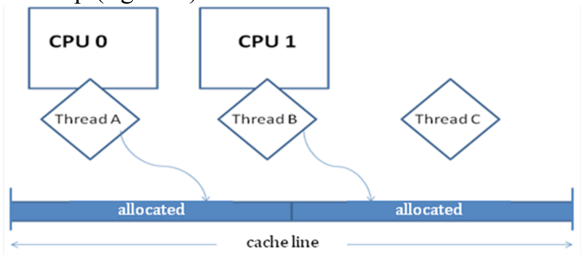

Fig. 1. two threads accessing the same cache line

This fake cache dividing line leads to serious performance problems. One method for solving this kind of problems is locking of a cache line by the thread which manipulates data contained in it at given time [3]. So, when a processor wants to update a cache line, first it must make all copies of this line which are located in caches of the other processors unavailable to them. When permitted, this processor can make a safe data update.

Still, a problem occurs when one cache line is continuously updated by a different processors and when the actual value of cache line is rapidly changing from one value to another. This behaviour makes different processors read and edit a single, certain cache line in short time intervals, so the problem that comes up is that a value requested by a processor may not be available when it is needed. This leads to (so called) cache-miss and the main memory must be approached, and that is not preferred. This is called cache sloshing [6].

Solving techniques for cache sloshing include using multiple CPU for allocation and thread allocation over hash arena which contains thread detectors (IDs). In practice, this aspect is satisfying so it became a part of various implementations of a malloc function. In *jemalloc*, multiple arenas are used, but a dedicated hash arena is not used [2].

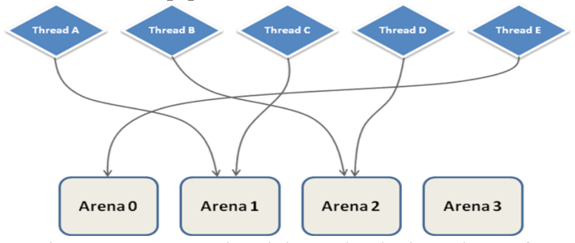

Fig. 2. Larson and Krishnan hash thread IDs for permanent thread assignment to arena. This is pseudo-random process, and therefore gives no guarantee that arenas will be used uniformly.

Range coding is a data compression scheme that was defined in 1979 by G. N. N. Martin [7]. Algorithm was desinged with aim to remove redundancy from a message in a digital form. Range coding is mathematically equivalent to arithmetic coding, technique which is, in contrast to range coding, encumbered with various patents. Therefore, range coding is widely used in open source communities, and telecommunication engineering.

The central concept behind range encoding is that if we are given a large-enough range of integers and a probability estimation for the symbols, the initial range can be divided into sub-ranges with sizes that are proportional to the probability of the symbol they represent appearing. Each symbol of a message can then be encoded in turns, by reducing the current range down to just that sub-range which corresponds to the next symbol to be encoded. Decoder must have access to the same probability estimation that encoder used, which means that it can either be sent in advance, derived from already transferred data or, most usually, be integral part of both encoder and decoder.

II. Benchmark Configuration

To test and compare *jemalloc* performance with *dlmalloc* library, the program *ebizzy* was used. This is an open-source application designed to simulate load of common web application. Application makes great use of threading and there are large demands for memory, so it very often allocates and deallocates working memory and copies it concurrently between threads.

In second part of benchmarking, free (as in freedom) range encoder and decoder by Michael Schindler were used. Source was slightly modified to make it easier to use external memory allocation libraries.

All tests were performed under the FreeBSD operating system version 7.2, compiled for 64-bit systems. Processor of the test configuration was Core 2 Duo ULV SL7100 1.2 GHz with 4MB of L2 memory. Configuration also had 2GB of DDR2 @ 667MHz.

In first benchmark suite, external library was used instead kernel version of memcpy function. This was done to eliminate advantages that jemalloc has because its integration in kernel. To stimulate memory fragmentation, memory locations were accessed randomly. In order to make better use of random memory chunks, binary search was used instead of linear.

Each of the benchmarks was run two times, first time with one physical core disabled and the second time with both cores enabled. The following block sizes were used for allocations: 256kB, 512 kB, 1 MB. Each of benchmarks was run for 10 seconds and resulting score is number of records processed per second. Record processing consists of reading from database, data processing and its re-entry into the database. In addition, 1, 2, 4, 8 and 16 threads were used by each of the benchmarks.

In second benchmark suite, both jemalloc and dlmalloc versions of range encoder were fed with random data ten times, mean time was taken, and difference in runtime was measured, along with mean deviation. After that, encoder output was decoded by decoders compiled against different libraries, and again, difference and deviation was measured. Process was repeated three times, with encoder input being 200 MB, 400 MB and 600 MB each time.

III. Benchmark and result evaluation

Results of first benchmark suite are presented in form of charts. First chart shows the results of benchmark when both cores of the processor are available, while on the second chart, results when just one core was available can be found. In both cases, the allocated blocks were 256 KB large.

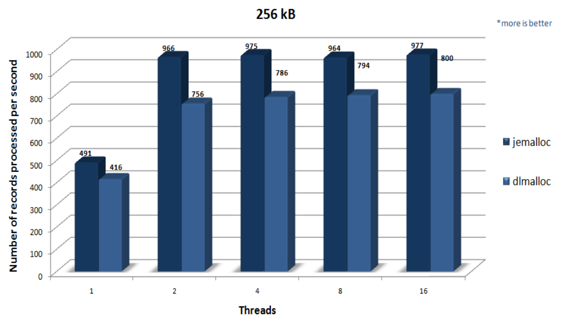

Chart 1: results of benchmarks while using a dual core processor at 256 KB block size

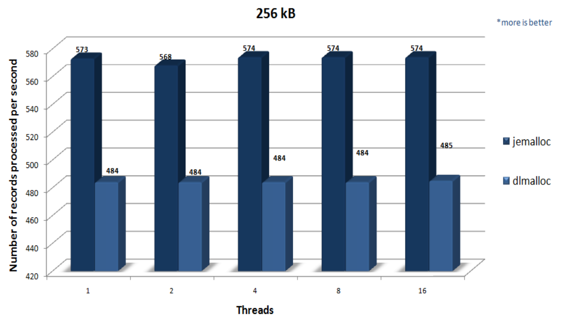

Chart 2: results of benchmarks while using a dual core processor, with one available core at 256 KB block size

As it can be seen on both charts above, ebizzy was processing faster on a processor with one core, while just one thread was running. This can be explained with scale of competition for resources when both cores are running. In case of using further threads, processing was about 40% faster than on just one core. Compared to dlmalloc, jemalloc profited a bit more from more threads.

Third chart shows the results of benchmark when both cores of the processor are available, while on the fourth chart, results when just one core was available can be found. In both cases, the allocated blocks were 512 KB large.

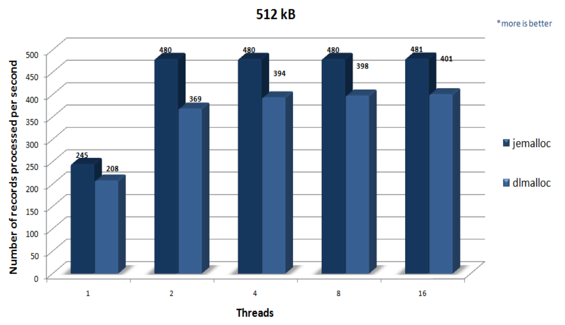

Chart 3: results of benchmarks while using a dual core processor at 512 KB block size

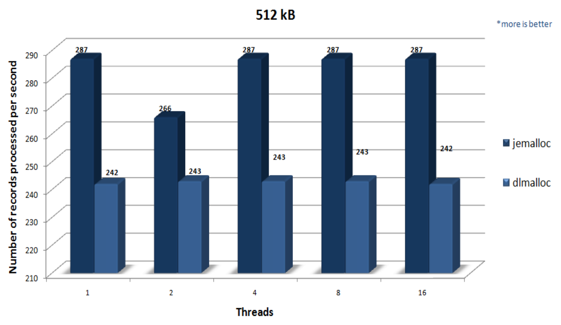

Chart 4: results of benchmarks while using a dual core processor, with one available core at 512 KB block size

Situation is very similar as the one with smaller block sizes. Again, compared to dlmalloc, jemalloc profited more from more threads. In this case, like in the former, performance is not negatively affected when number of threads is much larger than number of cores (which may in case of ebizzy be seen as an elevated number of clients who are trying to access the database).

Fifth chart shows the results of benchmark when both cores of the processor are available, while on the sixth chart, results when just one core was available may be found. In both cases, the allocated blocks were 1 MB large.

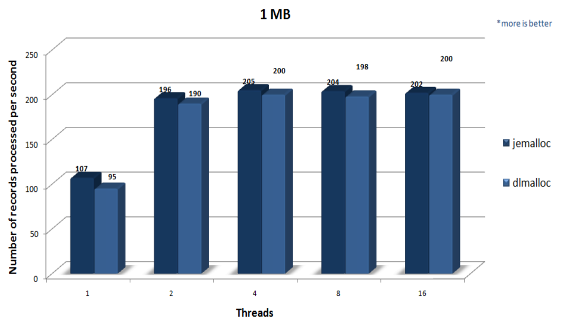

Chart 5: results of benchmarks while using a dual core processor at 1 MB block size

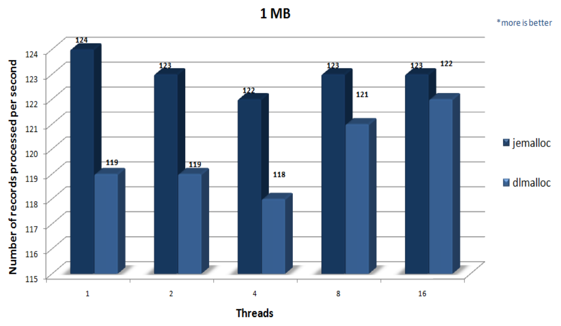

Chart 6: results of benchmarks while using a dual core processor, with one available core at 1 MB block size

While number of threads is not larger than number of cores, situation is still similar, jemalloc scales slightly better. However, once 4 or more threads are used, difference becomes smaller.

Results of second benchmark suite are shown in two charts. Seventh chart is showing difference between encoders, while eighth shows how well decoders compare, when compiled against different dynamic memory allocation libraries.

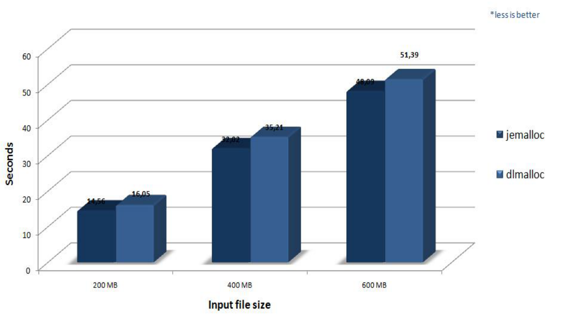

Chart 7: results of encoder benchmarks

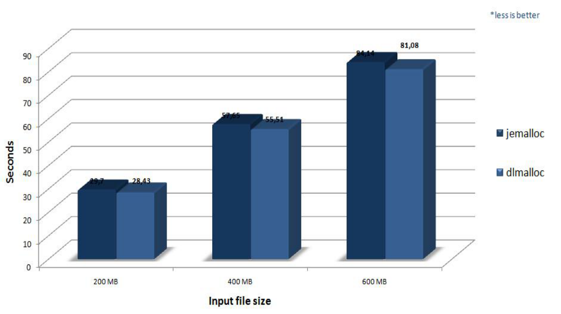

Chart 8: results of decoder benchmarks

From these two graphs, we can see that encoder is profting from using *jemalloc*, while decoder behaves better when compiled against *dlmalloc.*

IV. Conclusion

For the purpose of this paper, ebizzy software was used to benchmark improvements of jemalloc library, compared to the previously used library, dlmalloc. Performance of memory allocator is highly sensitive to application's allocation patterns, so definite conclusion can hardly be drawn. It is difficult to predict all possible allocation patterns that specific application can use, but simulators like ebizzy is can give very good overview of performance, due to their versatility. This was done to give some understanding of interns of dynamic memory allocators, so we can have a clear understanding why range encoder or decoder preforms better under certain conditions, if it does.

Side conclusion from this paper is that jemalloc should be used in applications that make heavy use of threading, and preferably use smaller memory blocks. If application uses larger memory blocks, at least 1 MB, other solutions should be investigated, because results are too close to say that jemalloc is strictly better, something else might perform better depending on an application's behaviour. The used benchmarks are also showing that performances of multi-threaded applications are properly scaling with number of processors, and, at the same time, it can be seen that performance of single-threaded allocation hasn't degraded.

Main conclusion is that jemalloc should be used for range encoders, and dlmalloc for decoders. Jemalloc implementation has better cache locality, and that aids a lot in case of encoder, due to nature of algorithm, which requires a lot of subranges, and lot of memory jumps, while manipulating data. However, we can say that dlmalloc is definitelly more versatile than jemalloc, and therefore, it suits decoder better, because algorithm is much more straightforward.

References

1) Mamagkakis S, Baloukas C, Atienza D, Catthoor F, Soudris D, Mendias J M, "Reducing Memory Fragmentation with Performance-optimized Dynamic Memory Allocators in Network Applications", Elsevier, 2005.
2) Jason Evans, "A Scalable Concurrent malloc Implementation for FreeBSD", FreeBSD.org, 2005.
3) Larson P, Krishnan M, "Memory allocation for long-running server applications", International Symposium on Memory Management (ISSM), 176–185, 1998
4) Altmeyer S, Burguiere C, "A new Notion of Useful Cache Block to Improve the Bounds of Cache-Related Preemption Delay", ECRTS Dublin, 2009
5) Frantisek Franek, "Memory as a Programming Concept in C and C++", Cambridge University Press, November 17, 2003
6) Kadayif I, Kandemir M, Chen G, "Studying interactions between prefetching and cache line turnoff", Shangai, 2005.
7) G. N. N. Martin, "Range encoding: an algorithm for removing redundancy from a digitized message", 1979