# High Performance Linux

High Performance Multi-core Networked and Storage Systems for Linux

## Tempesta FW

Open source Linux Application Delivery Controller

Wednesday, September 30, 2015

## Fast Memory Pool Allocators: Boost, Nginx & Tempesta FW

### Memory Pools and Region-based Memory Management

Memory pools and region-based memory management allow you to improve your program performance by avoiding unnecessary memory freeing calls. Moreover, pure memory pools gain even more performance due to simpler internal mechanisms. The techniques are widely used in Web servers and using them you can do following (pseudo code for some imaginary Web server):

```
http_req->pool = pool_create();
while (read_request & parse_request) {
    http_req->hdr[i] = pool_alloc(http_req->pool);
    // Do other stuff, don't free allocated memory.
}
// Send the request.
// Now destroy all allocated items at once.
pool_destroy(http_req->pool);
```

This reduces number of memory allocator calls, simplifies its mechanisms (e.g. since you don't free allocated memory chunks it doesn't need to care about memory fragmentation and many other problems which common memory allocators must solve at relatively high cost) and makes the program run faster.

Probably, you've noticed that we call *pool_alloc()* in the example above without specifying allocation size. And here we go to the difference between memory pools and region-based memory management: memory pools can allocate memory chunks with only one fixed size while region-based memory allocators are able to allocate chunks with different size. Meantime, both of them allow you not to care about freeing memory and just drop all the allocated memory chunks at once.

Boost C++ library provides memory pool library boost::pool, let's have a closer look at the library. By the way, there is nice description of memory pool concepts.

### Boost::pool

If we run following loop for N = 20,000,000 (this is body of a template function with argument *T*, you can find the full source code of the benchmark at our repository at GitHub):

```
boost::object_pool<T> p;

for (size_t i = 0; i < N * sizeof(Small) / sizeof(T); ++i) {
    T *o = p.malloc();
    touch_obj(o);
}
```

, it takes about 260ms for *T* 24 bytes (this is the size of structure *Small*) in size and about 170ms for an object of size 240 bytes (size of structure *Big*) on my Intel Core i7-4650U 1.70GHz. Please, note that I don't measure pools construction and destruction times to concentrate on memory management techniques without additional expenses. Also I limit number of allocations by

```
N * sizeof(Small) / sizeof(T)
```

to avoid memory exhausting on tests with large object sizes.

Google+

About Me

Alexander Krizhanovsky

CEO and founder of Tempesta Technologies and NatSys Lab.

View my complete profile

Popular Posts

Lock-free Multi-producer Multi-consumer Queue on Ring Buffer

Linux: scaling softirq among many CPU cores

Fast Memory Pool Allocators: Boost, Nginx & Tempesta FW

Fast Finite State Machine for HTTP Parsing

Studying Intel TSX Performance

NatSys Lock-free Queue vs LMAX Disruptor vs boost::lockfree::queue

How To Analyze Performance of Your Program

What's Wrong With Sockets Performance And How to Fix It

Haswell AVX2 for Simple Integer Bitwise Operations

Lock-free Condition Wait for Lock-free Multi-producer Multi-consumer Queue

Blog Archive

► 2016 (4)

▼ 2015 (2)

▼ September (1)

Fast Memory Pool Allocators: Boost, Nginx & Tempes...

► March (1)

► 2014 (3)

► 2013 (7)

► 2012 (9)

► 2011 (4)

Followers

Hereafter touch_obj() is just a simple macro checking result of memory allocation and breaking current loop:

```
#define touch_obj(o)                                    \
    if (__builtin_expect(!o, 0)) {                      \
            std::cerr << "failed alloc" << std::endl;   \
            break;                                      \
    } else {                                            \
            *(long *)o = 1;                             \
    }
```

Now if we free each 4th allocated element:

```
for (size_t i = 0; i < N * sizeof(Small) / sizeof(T); ++i) {
    T *o = p.malloc();
    touch_obj(o);
    if (__builtin_expect(!(i & 3), 0))
        p.free(o);
}
```

the function takes about 165ms and 155ms for *T* sizes 24 bytes and 240 bytes correspondingly and this is almost nothing for big objects, but more than 50% performance improvement for small objects. Results for the boost::pool benchmark:

```
$ ./a.out
             small object size:    24
               big object size:    240

           boost::pool (Small):    258ms
   boost::pool w/ free (Small):    165ms
             boost::pool (Big):    174ms
     boost::pool w/ free (Big):    154ms
```

(I take the best numbers from 3 runs of the test). The loop now contains extra conditional jump and *free()* call which is surely not for free. So why do we get the performance improvement for the version with objects freeing? There are two reasons why it's faster:

1. the main reason that in the first version of program we're allocating more than 450MB of RAM, so we cause serious pressure on system memory allocator which must allocate a lot of memory for us. While the second version of program requires 25% less system memory;
2. each write operation touches a new memory region, so some cache lines are evicted from CPU cache and the memory is loaded to cache. Thus, we get serious CPU cache starvation.

Therefore, periodic or casual memory freeings reduce overall workflow (while we have more *free()* calls, the underlying system allocator is called rarely) and effectively reuse CPU cache lines.

But why does big allocations test not win from freeing? The short answer is exponential growing of memory storage used by boost::pool and particular numbers (*N* and *sizeof(T)*)in the test. The key function of the boost::object_pool is pool<UserAllocator>::ordered_malloc_need_resize() (simplified for brevity):

```
template <typename UserAllocator>
void *
pool<UserAllocator>::ordered_malloc_need_resize()
{
    size_type POD_size = next_size * partition_size + ...
    char * ptr = (UserAllocator::malloc)(POD_size);
    ....
    if (!max_size)
        next_size <<= 1;
    ....
}
```

The function requests system allocator by exponentially growing steps. If we call *strace -e trace=mmap* against the test for small objects without freeing, then we can see following *mmap()* calls (glibc uses *mmap()* for large allocations and *brk()* for smaller):

```
mmap(NULL, 200704,....) = 0x7fdb428bd000
mmap(NULL, 397312, ...) = 0x7fdb4285c000
mmap(NULL, 790528, ...) = 0x7fdb4279b000
```

```
.....
mmap(NULL, 201330688, ....) = 0x7fdb295e3000
mmap(NULL, 402657280, ....) = 0x7fdb115e2000
```

12 calls in total for the benchmark code itself. The benchmark for small objects with freeing produces 11 calls ending at

```
mmap(NULL, 201330688, ....) = 0x7f91ed60b000
```

So the freeing really reduces size of allocated memory and improves performance. However, this is not the case for big allocation benchmarks - both the benchmarks have the same number of equal *mmaps()*:

```
mmap(NULL, 249856, ....) = 0x7f34431be000
....
mmap(NULL, 251662336, ....) = 0x7f3423f50000
```

I.e. in this particular test the last very large memory allocation is enough to satisfy the benchmark memory requirements even without freeing.

Boost::pool is the pure memory pool which allows you to allocate memory chunks of one fixed size only, but this is not the case for Nginx memory pool which is able to allocate objects of different size.

## Nginx Pool

Wikipedia says: *"the web server **Nginx**, use the term memory pool to refer to a group of variable-size allocations which can be later deallocated all at once. This is also known as a region"*. In fact, Nginx's pool allows you to allocate variable size memory chunks up to 4095 bytes for x86-64. You can request larger memory areas, but they're allocated using plain *malloc()*.

To learn performance of Nginx pool I copy & pasted some pieces of *src/core/ngx_palloc.c* from nginx-1.9.5. Now benchmark code for small and large memory allocations looks like the following (you can find the full code here):

```
ngx_pool_t *p = ngx_create_pool(PAGE_SIZE);

for (size_t i = 0; i < N * sizeof(Small) / sizeof(T); ++i) {
    T *o = (T *)ngx_palloc(p, sizeof(T));
    touch_obj(o);
}

ngx_destroy_pool(p);
```

Ngimx has *ngx_pfree()*, but it's for large data blocks only, so small memory chunks aren't freeable, so we test periodic memory freeings for large memory chunks only. Nginx's pool support different allocation sizes, so let's test mixed allocation with 1/4 of big (240B) and 1/4096 of huge (8305B) chunks:

```
ngx_pool_t *p = ngx_create_pool(PAGE_SIZE);

for (size_t i = 0; i < N; ++i) {
    if (__builtin_expect(!(i & 3), 0)) {
        if (__builtin_expect(!(i & 0xfff), 0)) {
            Huge *o = (Huge *)ngx_palloc(p, sizeof(*o));
            touch_obj(o);
            if (!(i & 1))
                ngx_pfree(p, o);
        }
        else if (__builtin_expect(!(i & 3), 0)) {
            Big *o = (Big *)ngx_palloc(p, sizeof(*o));
            touch_obj(o);
        }
        else {
            Small *o = (Small *)ngx_palloc(p, sizeof(*o));
            touch_obj(o);
        }
}

ngx_destroy_pool(p);
```

The code above gives us following results:

```
                ngx_pool (Small):    305ms
                  ngx_pool (Big):    132ms
           ngx_pool w/ free (Mix):    542ms
```

Let's do review of Nginx pool implementation and quick analyzing of the results:

1. the implementation is very simple (the ported allocator code takes less than 200 lines) without complex OOP wrapping code which boost::pool has in plenty;
2. *ngx_palloc()* scans each memory block at most 4 times trying to better utilize allocated memory chunks. There is small multiplication factor, but the algorithm is still O(1). If you do a small allocation after few large allocations, then it's likely that there is some room to place the new chunk somewhere in previously allocated blocks without requesting a new memory block. This is the cost for ability to allocate memory chunks of different sizes. Boost::pool just puts current memory chunk at the end of allocated data, so doesn't do any scans at all;
3. Nginx grows its pool by blocks of fixed size, it doesn't use exponential growing like boost::pool.

**Update.** Previously I explained the difference between results for *Big* and *Small* allocations by O(n) complexity in *ngx_palloc()*, in particular I was confused by this code:

```
do {
    m = ngx_align_ptr(p->d.last, NGX_ALIGNMENT);
    if ((size_t)(p->d.end - m) >= size) {
        p->d.last = m + size;
        return m;
    }
    p = p->d.next;
} while (p);
```

However, *ngx_palloc_block()* does

```
for (p = pool->current; p->d.next; p = p->d.next) {
    if (p->d.failed++ > 4)
        pool->current = p->d.next;
}
```

So when a new block is allocated the rest of the blocks increment *failed* counter, it means that we could not satisfy current request from all of them. Thus, we try to allocate a chunk at most 4 times from a block after that we move pool->current and never touche the head of the list. So typically we have very long head of the list and relatively small tail which is scanned in *ngx_palloc()*.

Thanks to Maxim Dunin and Rinat Ibragimov for pointing me out the mistake. Meantime, Rinat tried following change for *ngx_palloc_block()*:

```
- for (p = pool->current; p->d.next; p = p->d.next) {
-  if (p->d.failed++ > 4)
-   pool->current = p->d.next;
- }
+ pool->current = p_new;
```

I.e. he removed the scans at all. After the change benchmark results at his machine changed from

```
                ngx_pool (Small):    475ms
                  ngx_pool (Big):    198ms
           ngx_pool w/ free (Mix):    817ms
            ngx_pool cr. & destr.:    161ms
```

to

```
                ngx_pool (Small):    249ms
                  ngx_pool (Big):    163ms
           ngx_pool w/ free (Mix):    585ms
            ngx_pool cr. & destr.:    476ms
```

So the scans really degrades performance in particular workload from the benchmark, however strictly speaking the algorithm is still constant time.

The results are somewhat worse than boost::pool. Does it mean that boost::pool is better? Nope. Nginx is developed to process thousands or even more HTTP requests per second and each request has

associated with it memory pool. Obviously, regular HTTP request doesn't have 20 million chunks, headers or so on, so our test is somewhat unnatural for Nginx's pool. To compare Nginx's pool with boost::pool in more natural for Nginx workload I did other benchmark:

```
for (size_t i = 0; i < N / 100; ++i) {
    ngx_pool_t *p = ngx_create_pool(PAGE_SIZE);
    for (size_t j = 0; j < 100; ++j) {
        if (__builtin_expect(!(i & 3), 0)) {
            Big *o = (Big *)ngx_palloc(p, sizeof(*o));
            touch_obj(o);
        } else {
            Small *o = (Small *)ngx_palloc(p, sizeof(*o));
            touch_obj(o);
        }
    }
    ngx_destroy_pool(p);
}
```

Since boost::pool can't allocate various size chunks, I just ran 1/4 of all loops for *Big* and 3/4 for *Small* object allocations. The results are:

```
boost::pool cr. & destr.:    133ms
   ngx_pool cr. & destr.:    98ms
```

And this time Nginx pool is more than 30% faster than boost::pool.

By the way, APR library used in Apache HTTPD server uses somewhat similar to Nginx pool mechanisms.

## TfwPool

In Tempesta FW we tried to make more flexible pool allocator (actually region-based memory allocator, but we follow Apache HTTPD and Nginx developers in the allocator's name), such that it's able to free allocated memory like boost::pool and allocate variable size memory chunks like Nginx. But make it as fast as possible.

TfwPool uses **stack-like freeing** approach. In fact, there are two common cases when we need to free or reallocate the last allocated chunks. The first one is very trivial - when you just need a temporal buffer to do some stuff (e.g. *snprintf()*).

For example, checking of very short string with HTTP Host header looks like:

```
buf = tfw_pool_alloc(req->pool, len);
tfw_str_to_cstr(&field, buf, len);
if (!tfw_addr_pton(buf, &addr))
    // Process the bad Host header
tfw_pool_free(req->pool, buf, len);
```

*tfw_str_to_cstr()* copies chunked string to continuous *buf*. If you design your application as pure zero-copy, then typically such copies are not what you like to do. However, zero-copy techniques could be very expensive for small chunked data, when you need to handle list of small data chunks. So sometimes it's faster to do small copying rather than do complex things on lists of small chunks. Nevertheless, tfw_str_to_cstr() is considered deprecated to be replaced with smarter techniques in further versions.

The second case when you typically need to free exactly the last allocated memory chunk, is *realloc()*. Imagine that you need to handle some descriptor of string chunks of HTTP message headers, body etc. The descriptor is effectively an array of pointers to chunks of the message field. Arrays are better than linked lists because of better memory locality, but you must pay by heavy *realloc()* call to be able to grow an array. So if we'd have some very fast method to quickly grow an array, then we can reach very good performance. And there is a way to grow arrays quickly without usual for *realloc()* memory copies. It worth to mention that if you're reading for example HTTP message body, then you're growing the array of the field descriptor, while other allocated chunks are not touched. So at some point you just execute *realloc()* many times against the same memory chunk. If the allocator has some room just after the memory chunk, then it just need to move its data end pointer forward - no any real reallocations or copies are necessary.

Probably you noticed in previous code examples that I call free() just after alloc() - actually, this is relatively frequent case which we kept in mind designing TfwPool.

Like Nginx case I just copied original code with small adjustments to the benchmark. You can find

the original allocator code at GitHub. The results for the allocator are depicted at the below:

```
             tfw_pool (Small):    279ms
      tfw_pool w/ free (Small):    101ms
               tfw_pool (Big):    106ms
        tfw_pool w/ free (Big):    50ms
        tfw_pool w/ free (Mix):    107ms
         tfw_pool cr. & destr.:    53ms
```

Well, it's faster than Nginx's pool, but Small test is still slower than fro boost::pool. Surely, boost::pool shows amazing numbers thanks to it's exponential growing. We don't use the technique because our allocator was designed for Linux kernel and using large continuous allocations is not the most efficient way.

Our allocator is designed to operate with 4096-aligned memory pages. Page alignment (x86-64 has pages equal to 4096 bytes) is good since less pages are used improving TLB cache usage. In our benchmark we do the allocations using posix_memalign() with alignment argument 4096 and it significantly hurts the benchmark performance...

## Perfomance Issue with posix_memalign(3)

To understand *posix_memalign()* issues lets run memalign test which compares performance of 4KB allocations aligned to 16 and 4096 bytes correspondingly:

```
$ ./memalign_benchmark
                  16B aligned:    110ms
                 page aligned:    203ms
```

Page aligned allocations are almost 2 times slower, that's too expensive. Why it happens?

*strace* shows that more than 90% of time the program spends in *brk()* system call:

```
$ strace -c ./memalign_benchmark
                  16B aligned:    233ms
                 page aligned:    334ms
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 93.81    0.010849           1     17849           brk
  1.98    0.000229          13        17           mmap
  1.03    0.000119          12        10           mprotect
..............
```

If we run each part of benchmark independently, then we can find follwoing (firstly I run 16 bytes alignment test and next page alignment test):

```
$ strace -e trace=brk ./memalign_benchmark 2>&1 |wc -l
6087

$ strace -e trace=brk ./memalign_benchmark 2>&1 |wc -l
11769
```

The bigger number of *brk()* calls is expected since glibc requests more memory from the system to throw almost half of it for alignment. So large alignment argument is very expensive for *posix_memalign()*.

Meantime, operating system kernel manages memory in pages which are certainly 4096-byte aligned, so you can get properly aligned memory chunks without the alignment overhead. Moreover, actually when you do *malloc()* or *posix_memalign()*, the kernel must allocate VMA (Virtual Memory Area - the kernel descriptor handling information about address mapping) for it. Next when you write to the area page fault happens, it allocates a new page and populates it to current page table. A lot of things happen when we work with memory in user space. Then nice thing which glibc does to mitigate impact of the complex things is caching memory requested from the kernel.

## Tempesta FW: Native Kernel Implementation

It was also interesting to see how much a regular application can gain from moving to kernel space. First, to demonstrate how user space memory allocation is slow (if the application just starts and glibc doesn't have cache), I made very simple test:

```
static const size_t N = 100 * 1000;
```

```
static void *p_arr[N];

for (size_t i = 0; i < N; ++i) {
    r = posix_memalign((void **)&p_arr[i], 16, PAGE_SIZE);
    touch_obj(p_arr[i]);
}
for (size_t i = 0; i < N; ++i)
    free(p_arr[i]);
```

It takes about 112ms on my system:

```
$ ./memalign_benchmark
                  alloc & free:    112ms
```

Now let's try the kernel test, you can find it near from the other tests. You can run the test by loading the kernel module (it's built by Makefile as other benchmarks) and *dmesg* will show you the results (don't forget to remove the module):

```
$ sudo insmod pool.ko
$ sudo rmmod pool
$ lsmod |grep pool
```

Firstly, let's have a look how memory allocation is fast in kernel:

```
$ dmesg |grep 'alloc & free'
alloc & free:  22ms
```

This is **5 times faster**!

It's very curious how much regular application can gain from moving to kernel. So I also ported the Nginx's pool to kernel module and here are the results:

```
ngx_pool (Small):               229ms
ngx_pool (Big):                 49ms
ngx_pool w/ free (Mix):         291ms
ngx_pool cr. & destr.:          150ms
```

This is 30% faster for small allocations, 270% (!) faster for big objects and 86% faster for mixing workload... Wait... What's happen with our "real life" test for many pool with low number of allocations? It's slower more than 50%!..

The problem is that when you free a page using *free_pages()* interface some of them are going to per CPU page cache, but the main point is that they're going to buddy allocator where they can be coalesced with their buddies to satisfy further requests for large allocations. Meantime, glibc does good job on aggressive caching of allocated memory. So the benchmark causes significant pressure on Linux buddy system.

TfwPool uses it's own lightweight per-CPU page cache, so we get much better results:

```
tfw_pool (Small):               95ms
tfw_pool w/ free (Small):       90ms
tfw_pool (Big):                 45ms
tfw_pool w/ free (Big):         35ms
tfw_pool w/ free (Mix):         99ms
tfw_pool cr. & destr.:          54ms
```

In short: **80-170% faster** for small allocations than the fastest user-space allocator (boost::pool) and more than **3 times faster** than Nginx's pool,**3-5 times faster** for big allocations, more than **5 times** for mixed and **2 times** for "real life" workloads in comparison with Nginx. Recall that the allocator was designed for kernel space, so it show much better results in Linus kernel rather than for user space.

## Conclusion

As a conclusion I'd make couple of points about fast memory pools:

- while page aligned memory operations are good due to better TLB usage, they are impractical due to poor *posix_memalign()* work. You can use your own page cache to cope with the issue;
- exponential growing is good and old thing, but people still don't use it for some reason;

- any memory allocator must solve many problems (time complexity, memory utilization and fragmentation, time to create and destroy the pool etc.), so it's likely that you can develop much faster allocator than any existing for your particular project - there is no ideal memory allocator for all tasks;
- just moving from user space to kernel can make some performance improvement for your program, but also you can hit performance degradation - user-space and kernel use very different mechanisms, so kernel applications must be designed especially for kernel space.

Don't you love kernel programming as we love it?

## Update

Rinat Ibragimov added benchmark for Glib memory pool. Now the output of benchmark looks like (the best numbers for 4 runs):

```
                small object size:    24
                  big object size:    240
                 huge object size:    8305

             mallocfree (Small):    711ms
    mallocfree w/ free (Small):    478ms
              mallocfree (Big):    319ms
      mallocfree w/ free (Big):    204ms

            GLib slices (Small):    943ms
    GLib slices w/ free (Small):    621ms
              GLib slices (Big):    227ms
      GLib slices w/ free (Big):    111ms

            boost::pool (Small):    262ms
    boost::pool w/ free (Small):    169ms
              boost::pool (Big):    181ms
      boost::pool w/ free (Big):    157ms
       boost::pool cr. & destr.:    136ms

              ngx_pool (Small):    312ms
                ngx_pool (Big):    134ms
         ngx_pool w/ free (Mix):    554ms
          ngx_pool cr. & destr.:    99ms

              tfw_pool (Small):    273ms
       tfw_pool w/ free (Small):    102ms
                tfw_pool (Big):    107ms
         tfw_pool w/ free (Big):    46ms
         tfw_pool w/ free (Mix):    103ms
          tfw_pool cr. & destr.:    53ms
```

Posted by Alexander Krizhanovsky at 6:03 PM



# 10 comments:

**Roman Leventov** September 30, 2015 at 10:52 PM

After this talk http://www.youtube.com/watch?v=QBu2Ae8-8LM I became much more skeptical about 4KB pages, shifted my mind towards 2MB pages.

Reply

Replies

**Alexander Krizhanovsky** October 1, 2015 at 11:09 AM

Hi Roman,

thanks for the link - very interesting talk.

Huge pages are really good. However, there are few points about them good to have in mind:

1. huge pages are somwhat "unnatural" for hardware and OS. If the system runs for a while operating with common 4KB pages and now you're requesting 2MB page (or even 1GB!), then it could be hard for OS to find 2MB continuous memory area to satisfy your request. Hugetlbfs preallocates hugepages, so it should not hit the problem, while transparrent huge pages in Linux is typically sacrifices due to the issue;

2. some CPUs aren't able to support 1GB pages in virtualization mode, so you're limited by 2MB pages only in your VMs;

3. huge page translations are cached by separate TLB which is significantly smaller than the one for common pages. However, it still caches much more memory in total in comparison with common TLB.

Actually, we use huge pages in Tempesta DB (https://github.com/natsys/tempesta/blob/master/linux-3.10.10.patch#L590). However, we preallocate all of them at system boot time when there is plenty of free space.

**Roman Leventov** October 1, 2015 at 6:26 PM

Haswell has significant improvement - it adds L2 TLB for huge pages. Waiting for Skylake specs now.

**Pavel Odintsov** March 22, 2016 at 6:24 PM

Thanks for link! Very useful!

**Alexander Krizhanovsky** March 22, 2016 at 7:36 PM

Pavel, you're very welcome ;)

**Reply**

**JimJag** October 2, 2015 at 9:51 PM

The Apache pool mechanism, which is now in APR has been in Apache since before Apache 1.0.0 and is likely the grandfather of Nginx's implementation.

Reply

**KjellKod** October 4, 2015 at 7:47 PM

Have you tried Google's perf tools, I.e. The memory allocator tcmalloc?

Reply

Replies

**Alexander Krizhanovsky** October 4, 2015 at 11:07 PM

Tcmalloc is generic allocator, while usually we develop task-specific allocators to gain more performance or leave with glibc for non-critical code.

**Reply**

**Pseudonym** October 5, 2015 at 1:44 AM

Just as a matter of curiosity, could you also try libumem? I used it a lot back when I was doing a lot of Solaris. I'm curious how it stacks up.

Reply

Replies

**Alexander Krizhanovsky** October 6, 2015 at 12:58 AM

Actually, there are too many of pool and region-based allocators to try them all, but I'll be happy to see any benchmarks from you about a new allocator.

**Reply**

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)