NUMA Allocators in Real-Time Multiprocessor Systems

Bhavesh Bareja
Computer Engineering
University of Michigan
Ann Arbor, USA
bhaveshb@umich.edu

Bradley Baker
Computer Engineering
University of Michigan
Ann Arbor, USA
bvbaker@umich.edu

Juechu Dong
Computer Engineering
University of Michigan
Ann Arbor, USA
joydong@umich.edu

Paul George
Computer Engineering
University of Michigan
Ann Arbor, USA
egpaul@umich.edu

Dino Mastropietro
Computer Engineering
University of Michigan
Ann Arbor, USA
donatom@umich.edu

***Abstract*—This project aims to compare, adapt, and explore alternatives to existing memory allocators for Non-Uniform Memory Access (NUMA) architectures within the context of soft real-time systems. We compared glibc's Malloc, TCMalloc and a modified TLSF where we attempted to add huge page allocation, hoping to reduce the number of page faults and TLB evictions. Our findings show that TCMalloc has a very short average latency for allocation but upon start up will have long latencies.**

I. Introduction

Non-Uniform Memory Access (NUMA) systems are becoming more ubiquitous as more sophisticated CPU-memory architectures are designed to support the growing memory channel count and core count. The cores experience lower memory access latencies for in-node access, while cross-node memory access has additional latency from inter-node connections. Allowing NUMA gives architects more flexibility on CPU-Memory architecture, for example it enables multi-socket design. However, the use of NUMA has been considered as a source of non-determinism in the real-time domain, since the problem of solving for worst-case memory access latency is intractable and the worst run-time is bounded by the worst-case memory latency.

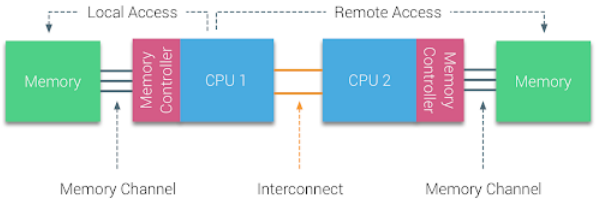

Fig. 1. A NUMA System

As opposed to traditional hard real-time systems where the deadline cannot fail under any circumstances, modern real-time tasks are often "soft" real-time, where tail latency instead of worst-case latency is the performance target. Unlike hard real-time systems, for example aviation computers where failing a deadline means total failure to meet the performance target, soft real-time applications like video games can take some failures as long as it doesn't happen too often. [1]

A common solution for multi-threaded real-time applications running on NUMA, occasionally even multicore real-time applications in general, is to isolate each node and statically schedule each process/thread to a node and statically place its corresponding memory to the same node. This strategy greatly limits CPU and memory utilization and throughput due to poor load balance. As modern real-time applications offer more bandwidth challenges rather than stricter deadlines, and more "soft" deadlines, it's worth reevaluating the pros and cons of this strategy. [2]

Popular open sources and widely used allocators are either not designed with real-time systems in mind, or do not work with multi-core systems. General purpose allocators like tc_malloc routinely reorder existing blocks to deallocate empty pages, resulting in non-constant allocation/deallocation latency. This feature increases memory utilization but fails to meet tail-latency requirements. Dynamic real-time allocators like tlsf guarantee constant allocation/deallocation time but are often designed with single-core, light-weight embedded systems in mind. On multi-core architectures, these allocators divide the system into multiple nodes, each having a dedicated core, memory, and memory management page. Each thread is scheduled to one particular core. This policy makes it hard to maintain load balance. Static real-time allocators based on profiling and workload analysis remain mainstream because of the poor performance of dynamic allocators on multicore systems, sacrificing flexibility. As a result, industry designs closed source, specialized allocators for their specific real-time application. [3] [4]

II. Allocators

In this section we explore the currently available Dynamic Storage Allocation (DSA) algorithms and asses their pros and cons. As we will see, most allocation algorithms, while they work well with the general workload of multi-threaded systems, are are either not optimized for the system and rely on OS to efficiently handle that part or are not optimized to tackle the hard requirements for real-time applications. As far as we have seen, there are no allocators optimized to handle

the applications running on NUMA architectures with Real time requirements.

A. PTmalloc

PTmalloc or pthread malloc is the base of the default allocator used by glibc. This algorithm subdivides the memory into multiple "Arenas" with one main arena or the initial heap of the program and multiple arenas to be used by threads. This is done to reduce contention among threads but ultimately results in larger fragmentation with the growing number of threads and unbounded delays as the contention among threads increases. While PTmalloc is aware of mutilple threads, there is no code to optimize it for NUMA architectures, coordinate thread locality, sort threads by core. It is assumed that the kernel will handle those issues sufficiently well [5]. In our study we found that this algorithm has a large overhead of managing the free block as it requested memory from kernel very frequently and freed the blocks as frequently too. This is might be near optimal for applications with limited memory, the overhead of releasing and requesting memory very often is not ideal for the latency

B. TCMalloc

TCmalloc or Thread Cache malloc is the name derived from its legacy implementation of its front-end "Per thread cache" mode where each thread had its own cache; however, this resulted in larger memory footprints that scaled with the number of threads. Now with the current implementation of per CPU mode, it assigns a cache to each logical CPU rather than each thread. The front-end is responsible for handling requests for memory by the application. Each cache can only be accessed by one thread at a time and hence does not require locks to handle contention. This greatly helps the latency and most requests are handled relatively fast as compared to PTmalloc which requires the use of locks. If the front end has exhausted its cache or needs a chunk larger than the one available, a request is sent to the middle end that constitutes the Transfer Cache and Central Free List. These structures help the transfer of memory between CPUs happen rather seamlessly. If the middle end also runs of space to be allocated or moved from another CPU, the back end requests the kernel for more space. To handle the whole process as efficiently as possible, the algorithm requests the OS for huge chunks of memory (about 1 GiB) at a time [6]. While this memory is only reserved and not backed by physical memory, the overhead of requesting such huge chunks makes its tail latency rather large an unreliable for real time applications.

C. TLSF

TLSF or Two Level Segregated Fit List is the best candidate for real time applications as it guarantees O(1) for allocation and was built with real time applications in mind. It relies on a few assumptions that make it fast but leave some security flaws unaddressed. The authors made the following assumptions while creating this allocator [7]:

- Trusted environment: programmers that have access to the system are not malicious, that is, they will not try to intentionally steal or corrupt application data. The protection is done at the end-user interface level, not at the programming level.
- Small amount of physical memory available.
- No special hardware (MMU) available to support virtual memory.
- No reallocation: It is assumed that the original memory pool is a single large block of free memory, and no sbrk()function is available.

Having these assumptions takes a lot of housekeeping away from the algorithms responsibility and hence has a huge impact in reducing latencies. The other reason that TLSF can guarantee O(1) execution is due to the algorithm used to manage the free memory blocks. It maintains two lists, the first level divides free blocks into classes that are a power of two apart (16, 32, 64, 128, etc) and the second-level sub-divides each first-level linearly. Each array of lists has an associated bitmap used to mark which lists contain which free blocks. To calculate the indexes given a block size, TLSF relies on an internal function, segregate_list() that returns the first and second level indexes for the requested block. This helps in bounded time execution for a search of the requested size of block. Another differentiator that sets TLSF apart from rest of the available allocators is that it employs active coalescing of blocks that are freed. To do this efficiently TLSF embeds each block with some extra information like:

- The size of the block, required to free it and to link this block with the next one in the physical link list.
- Boundary tag, a pointer to the head of the previous physical block.
- Two pointers, to link this block into the corresponding segregated lists.

Having active coalescing helps the algorithm maintain a very low level of fragmentation that ultimatley helps in better performance.
While it is able to guarantee O(1) execution, the assumptions with which it was written restrict it to not be used with NUMA architectures.

D. TLSF-hp

Since TLSF assumes that the initial pool of memory is all there is and does not expand its initial allocation, we theorized that it can be expanded efficiently to NUMA architectures by allocating HugePages to each thread. A HugePage as the Figure 2 shows consists of multiple normal pages. Prefaulting them to each thread will help in expanding the application of TLSF to NUMA architectures.
Unfortunately our implementation could not be seen through as the algorithm kept segfaulting with the t-test1 test bench from rpmalloc suite and we could not figure out the source of memory leak even after using tools like Valgrind, gdb etc. An interesting thing to note was that when used with Valgrind, the overhead somehow gave the algorithm enough time to resolve

the memory leak. But that also meant that we could not profile the implementation correctly. We were also able to pass the basic sanity checks for our muti-threaded implementation of the same.

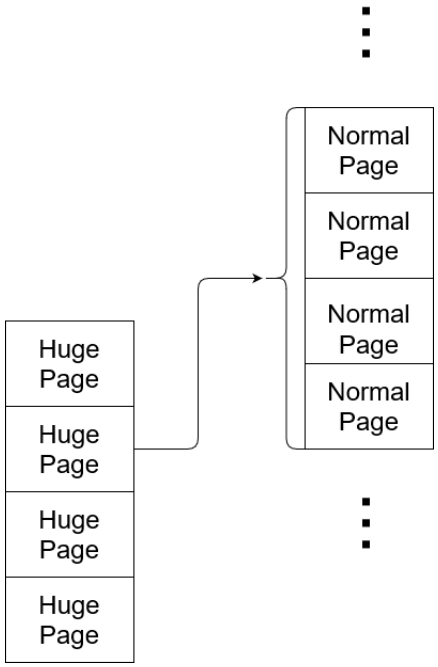

Fig. 2. Huge pages are page allocations from one level up in the virtual page table where allocating one huge page allocates all pages pointed to by that huge page

III. Tools and Test Setup

The main tool used in evaluating our allocators was cyclictest [8]. Cyclictest is a multithreaded program that runs a loop for a set amount of time at set intervals of timing. The interval set for each thread uses the system timer to wake up with whatever preemption priority is set by the programmer. At wake up, the thread grabs a timestamp, runs a task, and then grabs another timestamp to measure the length of that test. This lets us measure allocation latency in terms of average, minimum, and maximum, as well as a list of all allocation latencies. The cyclic nature of the program allows us to observe a few important factors in real-time systems, being preemption and stability. We can set the priority of cyclictest to test the effects of different background workloads and pre-emptions on top of cyclictest. We can also observe long runs of cyclictest as a stress test to check stability and observe the results of long periods of repeated, differently-sized allocations.

The test case running in cyclictest is a combination of a malloc test case from RTEMS [9] as well as a test case we came up with. The RTEMS malloc test was a fairly simple malloc stress tester. It just calls repeated small mallocs of random sizes within a small range and then frees them. The test case we came up with ended up being a similar test, but we allocated much larger pages and forced the allocations to remain allocated for a set number of rotations, allowing us to ensure that the stress test does more than allocating the same memory as well as giving us control over the total allocation active in the system.

A. PREEMPT_RT & Linux Low latency kernel optimizations

Preempt_RT is a fully preempt-able configuration of the Linux kernel. It creates an additional range of process priority classification (0-99, higher value higher priority) which allows it to preempt other process (even kernel processes) with a lower assigned priority. Beyond Kernel Side Preemption, support is enabled for High Resolution Timers, Priority-Inheritance (avoids priority inversion problem), Earliest-Deadline First scheduling and Real-time locks.

Subsystem	Priority	Description
Arm bL switcher	1	The Arm big.LITTLE switcher thread
crypto	50	Crypto engine worker thread
ACPI	1	ACPI processor aggregator driver
drbd	2	Distributed, replicated block device request handling
PSCI checker	99	PSCI firmware hotplug/suspend functionality checker
msm	16	MSM GPU driver
DRM	1	Direct rendering request scheduler
ivtv	99	Conexant cx23416/cx23415 MPEG encoder/decoder driver
mmc	1	MultiMediaCard drivers
cros_ec_spi	50	ChromeOS embedded controller SPI driver
powercap	50	"Powercap" idle-injection driver
powerclamp	50	Intel powerclamp thermal management subsystem
sc16is7xx	50	NXP SC16IS7xx serial port driver
watchdog	99	Watchdog timer driver subsystem
irq	50	Threaded interrupt handling
locktorture	99	Locking torture-testing module
rcuperf	1	Read-copy-update performance tester
rcutorture	1	Read-copy-update torture tester
sched/psi	1	Pressure-stall information data gathering

Fig. 3. Sample Kernel Subsystem Priority Levels

Using Preempt_RT requires building the Linux kernel from source and then booting a compatible distribution with it. The RHEL-8 High Performance and Low Latency Tuning Guide [10] outlined many different ways by which one could further reduce execution time variance with a Preempt_RT kernel, namely , by isolating cpu cores, disabling interrupt balancing, disabling power-saving cpu frequency governors, locking pages, pre-faulting pages, disabling the VGA console, and disabling transparent Huge pages (instead using Vanilla HugePages). We evaluated and enabled the above and were able significantly improve average as well as worst case latencies reported by cyclictest.

B. NUMA

NUMA or Non-Uniform Memory access arises out of the physical organization of the compute and memory resources of a system. Extracting performance from such systems requires careful data and process thread mapping so as to minimize data movement overheads and their associated complex implications for a processes worst case execution time. Enthusiast Consumer and High performance embedded controllers have been shipping with with NUMA configurations for the last decade.

We evaluated Gem5 [11] a popular simulator for systems research. Gem5 allows for a large variety of system configurations, esp for us NUMA configurations, at varying degrees

	glibc	tcmalloc
max	118403 ns	5576319 ns
median	3411 ns	3059 ns
IQR	370 ns	350 ns
mean	3355 ns	3110 ns
std dev	394 ns	387* ns
99th percentile	3915 ns	4066 ns
num high outliers	0.78%	1.60%
num extremes	0	7

Fig. 4. Key statistics from a 3-minute sample of cyclictest

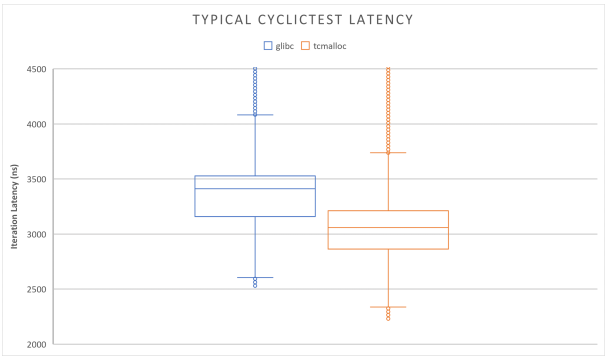

Fig. 5. Service distribution of cyclictest latency in glibc vs. tcmalloc

	glibc	tcmalloc
munmap	4.4M	33
mmap	4.4M	78
madvise	8	900k
brk	3	15

Fig. 6. strace call profile with cyclictest

of resolution(full system(configurable micro-architecture detail), and system call emulation mode) we determined that it would not be suitable for our memory heavy workloads in full system mode, and unsuitable for measuring run-time latencies in system call emulation mode. Approaches exist to virtualize(gem5-qemu) some parts of the gem5 full system execution and overcome these limitations but we were unable to bring it up with our desired configuration.

The Linux kernel allows for the partitioning of the underlying memory into Fake NUMA regions to enable the test and bringup of real numa systems, as well as a means to partition the system memory if isolation were desired. Enabling FakeNUMA at boot time gives us both a divided memory subsystem and no simulation overheads that impact its latency.

IV. Results

Unfortunately, we have been unable to get our TLSF implementation to run under the pressure of cyclictest. We have passed the test suite that TLSF uses, but we have been trying to trace segfaults that appear when we run TLSF at full speed. The issue disappears when we add debug statements or run Valgrind, which made it tough to trace the error properly. This means that our primary results are a display of the abilities of our testing setup and an indication of the important factors we would analyze in TLSF. The biggest factors we would like to check are the number of outliers and the average latency of malloc calls. Below is shown the results for glibc and tcmalloc run for 20 minutes with our test and with the RTEMS malloc test, as well as what we were hoping to achieve with TLSF.

There are a few interesting results from these tests. The first is that tcmalloc is a pool allocator, so we expect that the only extremes occur at the beginning of the program. The data we measured shows that the only extremes were indeed at startup. Beyond that, we see that tcmalloc and glibc perform quite similarly - neither have extremes during the normal run of cyclictest, and the 99th percentile performance are very similar. (*)The actual full-sample standard deviation of tcmalloc is skewed quite high due to how extreme the startup cost is: 12490 ns. Removing the startup cost shows us a measure of steady state tcmalloc operation, which is the value shown in the table. All measures of center and spread favor tcmalloc. This indicates that ignoring startup cost, tcmalloc appears to satisfy 99% of requests better than glibc. The number of high outliers is the category in which tcmalloc falls short, indicating that the distribution is less centered than the glibc distribution. Ideally, we could have used these metrics to characterize performance of TLSF in this environment and compare it against tcmalloc. The key factors TLSF would be attempting to win would be in spread and the number of outliers.

Figure 6 shows the stark contrast between glibc and tcmalloc's way of handing memory. As mentioned in Section III, glibc frequently requests and releases memory to OS, the same can be seen with strace call pattern. While glibc calls mmap and munmap about 4.4M times each, tcmalloc calls them not even for a fraction of times. Rather, it uses madvise to tranfer memory blocks between multiple threads/CPUs.

V. Challenges

Since dynamic memory is less common in real-time applications, we had difficulty finding some really strong testbenches that would stress memory allocation in a way that is relevant to real-time systems. Profiling the performance metrics such as reliable timing of allocators and fragmentation caused by them was another major challenge we faced.

VI. Future Direction

Currently, our TLSF implementation is not aware of NUMA nodes, instead using global pools and locks to manage subpools. We didn't have time to implement it, but we wanted to investigate the possibility of allocating huge pages for the 'global' pool on each NUMA node that needs an allocation. That pool would only be used by the CPU nearest its node

to minimize latency. This would add space overhead scaling linearly with the number of NUMA nodes as each node would need to initialize its global pool, but this would remove the need for locks and guarantee that memory access latency is minimized.

Additionally, since TLSF already assumes that it runs in a trusted environment, we would like to investigate the possibility of sharing huge pools of memory between applications on the same node. This could allow us to allocate even larger global pools since huge pool allocations would not take memory capacity away from other processes. This would limit the applicability of this allocator to extremely trusted environments, but we would be able to test whether we can turn application startup time into system startup time. This could allow new applications to avoid page faults on startup while still receiving a lot of the benefit of pre-allocated memory.

A. *Verification*

Most System level validation latency validation today is done by stressing combinations of subsets of the overall system and record their impact on the test time "worst" case execution times of the real time workload. This approach is unsuitable for hard real-time systems or for systems that have firm QoS guarantees. Statically analyzing Worst case execution times, would require quantifying all sources of latency and their inter-dependencies, and thereafter enumerating all possible execution traces, with all possible architecture / micro-architecture / operating system forced inter leavings. Efforts in the community [12] [13] have shown that this is tractable in finite-state (heavily isolated and predictable) systems, however scaling these to today's high performance micro-architectures and operating systems like those that made up our test platform is still an open problem.

VII. Contribution

1. Bhavesh: 20%
2. Brad: 20%
3. Juechu: 20%
4. Paul: 20%
5. Dino: 20%

References

[1] L. A. G. C. Buttazzo, G. Lipari and M. Caccamo.

[2] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *2011 IEEE 32nd Real-Time Systems Symposium*, 2011, pp. 217–226.

[3] V. Shah, "A review on memory allocators for numa based rtos," 06 2018. [Online]. Available: http://dspace.hmlibrary.ac.in:8080/jspui/bitstream/123456789/1366/18/18_Synopsis.pdf

[4] R. Gifford, N. Gandhi, L. T. X. Phan, and A. Haeberlen, "Dna: Dynamic resource allocation for soft real-time multicore systems," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021, pp. 196–209.

[5] [Online]. Available: https://sourceware.org/glibc/wiki/MallocInternals

[6] [Online]. Available: https://google.github.io/tcmalloc/design.html

[7] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "Tlsf: a new dynamic memory allocator for real-time systems," *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.*

[8] [Online]. Available: https://wiki.linuxfoundation.org/realtime/documentation/howto/tool

[9] [Online]. Available: https://devel.rtems.org/wiki/Developer/Git

[10] R. Inc., " Optimizing RHEL 8 for Real Time for low latency operation," https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/8/html-single/optimizing_rhel_8_for_real_time_for_low_latency_operation/, 2022.

[11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, aug 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[12] R. Cardell-Oliver, "The formal verification of hard real-time systems," Ph.D. dissertation, University of Cambridge, 1992.

[13] S. Campos, E. Clarke, W. Marrero, and M. Minea, "Verus: A tool for quantitative analysis of finite-state real-time systems," in *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, Tools for Real-Time Systems*, ser. LCTES '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 70–78. [Online]. Available: https://doi.org/10.1145/216636.216661