2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)

Ptlbmalloc2: Reducing TLB Shootdowns with High Memory Efficiency

Stijn Schildermans
Kris Aerts
Department of Computer Science
KU Leuven campus Diepenbeek
Wetenschapspark 27
3590 Diepenbeek, Belgium
{*stijn.schildermans*}{*kris.aerts*}@*kuleuven.be*

Jianchen Shan
Department of Computer Science
Hofstra University
Hempstead
New York 11549-1000
United States of America
jianchen.shan@hofstra.edu

Xiaoning Ding
Department of Computer Science
New Jersey Institute of Technology
University Heights
Newark, New Jersey 07102
United States of America
xiaoning.ding@njit.edu

***Abstract*—The cost of TLB consistency is steadily increasing as we evolve towards ever more parallel and consolidated systems. In many cases the application memory allocator is responsible for much of this cost. Existing allocators to our knowledge universally address this issue by sacrificing memory efficiency. This paper shows that such trade-offs are not necessary by presenting a novel memory allocator that exhibits both excellent memory efficiency and (TLB) scalability: *ptlbmalloc2*.**

First, we show that TLB consistency is becoming a major scalability bottleneck on modern systems. Next, we describe why existing memory allocators are unsatisfactory regarding this issue. Finally, we present and evaluate ptlbmalloc2, which has been implemeted as a library on top of glibc.

Ptlbmalloc2 outperforms glibc by up to 70% in terms of cycles and execution time with a negligible impact on memory efficiency for real-world workloads. These results provide a strong incentive to rethink memory allocator scalability in the current era of many-core NUMA systems and cloud computing.

1. Introduction

In x86, the translation lookaside buffer (TLB) is a per-core cache for page table entries (PTEs), allowing rapid translation of virtual to physical memory addresses. In contrast to regular caches, TLB consistency is implemented by the operating system (OS), using a mechanism based on inter-processor interrupts (IPIs); called a *TLB shootdown*. In contrast to historical dogma, recent work shows that this mechanism can be very costly, in particular in parallel or virtualized contexts [1]–[4]. Given that such environments are becoming ever more ubiquitous, reducing TLB shootdown overhead is paramount.

Improving TLB performance has been extensively studied, as discussed in §8. Solutions have been proposed at both hardware and system software levels. However, the former are expensive and have long adoption intervals, while the effectiveness of the latter is limited due to the semantic gap between the system and application, requiring efficiency to be sacrificed to guarantee correctness [5]. It is therefore evident that system-level solutions alone are insufficient.

While system software can optimize TLB-sensitive operations to a certain degree, their root cause often lies at application level. For multithreaded applications in particular memory management is often the main cause of TLB shootdowns [4]. Moreover, optimizations at application level may be much more efficient than those at system-level, since only there the exact contents of the virtual memory space are known. This is underpinned by the fact that many memory allocators in fact do not send many TLB shootdowns. However, this is merely a side effect of sacrificing memory efficiency in favor of other design goals such as efficient garbage collection, upon which we elaborate in §4.4. To our knowledge, no existing memory allocator combines high memory efficiency with excellent (TLB) scalability.

Given the above, it is evident that the balance between memory efficiency and (TLB) performance should be reconsidered in modern memory allocator design at a conceptual level. The main goal of this paper is to do exactly that, supported by pragmatic evidence. To achieve this, we first quantify the performance overhead of TLB consistency in modern systems, emphasizing how certain system properties impact said overhead. Next, we show how existing memory allocators (fail to) deal with this growing problem. From this knowledge we derive how this problem can be dealt with conceptually, based on a handful of heuristics. Next, we apply our concept in C, in the form of a novel memory manager implemented as a library on top of glibc: *ptlbmalloc2*. We conclude by evaluating the performance of ptlbmalloc2 compared to glibc's ptmalloc2 for real-life workloads.

1.1. Contributions

• We quantify TLB shootdown overhead with respect to several system properties and show this is a growing issue;
• We identify the *arena imbalance issue*, which may cause excessive TLB shootdowns in efficient memory allocators;
• We introduce an allocator design concept combining TLB scalability with memory efficiency: *global hysteresis*;
• We present and evaluate *ptlbmalloc2*: an implementation of our memory management concept as a C library.

978-1-6654-1485-2/20/$31.00 ©2020 IEEE
DOI 10.1109/ISPA-BDCloud-SocialCom-SustainCom51426.2020.00036

2. Background: TLB Consistency

As described in §1, the TLB is a cache for PTEs. Thus, whenever a PTE changes, the TLB must be updated. Many operations on virtual memory may result in said PTE changes. Some of these operations are initiated by the system (e.g. page migrations, memory compaction,...), while others are initiated by applications via system calls (`munmap`, `madvise`, `mprotect`,...) [6]. These system calls usually encompass returning virtual memory to the system or changing memory page permissions.

Because modern CPUs often use physically-tagged caches, the TLB is consulted on every memory access. It is thus performance-critical, and therefore local to each core. This means that TLB consistency issues arise in multi-core (SMP) systems. When a PTE changes, all cores buffering it in their TLB must be notified. The majority of CPU architectures, including x86, rely on the OS for this for apparently mainly historical reasons [7]. In Linux, this is done by flushing the PTE from the local TLB through a dedicated instruction and sending an IPI to all CPUs that use the virtual memory space containing the altered PTE, instructing them to do the same. On modern Intel CPUs (utilizing x2apic), each IPI is sent by writing the destination CPU and interrupt vector to the ICR MSR [8]. The local APIC then sends the IPI to the receiving CPU. The latter processes it and acknowledges its receipt by writing to the EOI MSR. The sending CPU synchronizes with all recipients by means of a spin-lock before proceeding. This entire process constitutes a TLB shootdown.

In virtualized systems, performing a TLB shootdown is significantly more complicated. Firstly, IPIs are in this context sent between virtual CPUs (vCPUs). Since the vCPU the guest OS sees is not guaranteed to be running on the corresponding physical CPU, the hypervisor (VMM) must intercept all guest IPIs. In systems employing hardware-assisted virtualization -the standard virtualization method today- the CPU itself intercepts IPIs by triggering a *VM exit* when the guest attempts to write to the ICR MSR, handing control to the VMM [8]. The latter will then reroute the IPI to the correct CPU. VM exits are expensive operations. Besides the direct execution of VMM code, their cost includes time to save/restore the guest state, as well as increased cache pressure [9]. On older systems, VM exits are required on the receiving CPU as well to process the interrupt. Recent Intel Xeon CPUs however contain an optimization (APICv) that eliminates these recipient-side VM exits [8]. Secondly, virtualization often abstracts the host system's NUMA layout from the guest, leading to an increase in slow cross-socket IPIs on NUMA systems since the guest scheduler can not optimize thread/memory placement. Thirdly, if multiple VMs share the same physical CPU set, a vCPU may send an IPI to a vCPU that is currently preempted. It must then wait until the receiving vCPU is re-scheduled and acknowledges receipt of the interrupt. This is known as *TLB shootdown preemption* [10]. Several optimizations exist to limit the impact of this problem [3], [8]. However, many systems are not yet equipped with these recent optimizations.

3. Rising TLB Shootdown Cost

Previous studies have already quantified the cost of TLB shootdowns at a low level [2], [11]. However, from these studies the real-world cost in realistic environments remains unclear. From a pragmatic perspective, this insight is crucial. In this section, we aim to provide such insight by evaluating TLB shootdown overhead in a variety of realistic scenarios, in function of several common system properties.

3.1. Experimental Setup

From system source code and literature, we find that three system properties may significantly influence the cost of TLB shootdowns: CPU count used by the workload, system NUMA architecture and virtualization [11], [12]. We will thus focus on these properties in our analysis.

Our test system is an Ubuntu 18.04 (kernel 4.15) Linux server with 4 Intel Xeon CPUs (20 cores, 40 threads each) and 256GB of RAM. For experiments requiring virtualization, we use KVM. We use `taskset`[1] to pin threads to the desired cores. We record the number of TLB shootdown IPIs, the number of CPU cycles consumed by the program, and wall-clock time using `perf`[2].

The test workload is a C program that creates 16 threads that each call `madvise(MADV_DONTNEED)` 1,000,000 times in a loop. As noted in §2, this is one of the system calls known to induce TLB shootdown IPIs. The source code of our test program is available on GitHub[3].

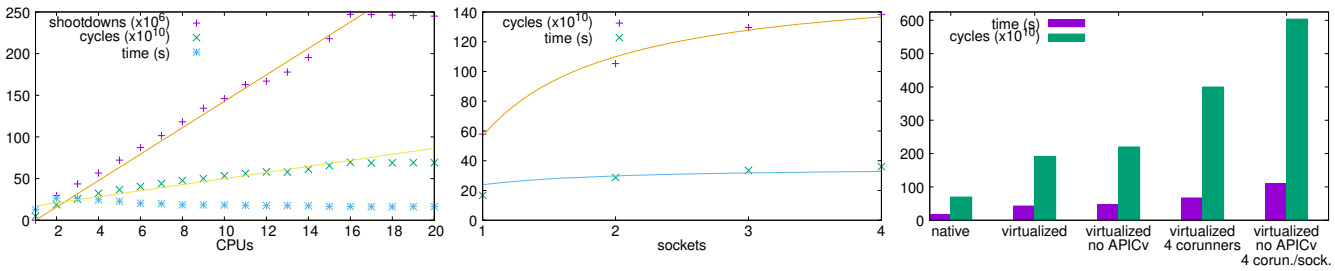
3.2. Results

3.2.1. CPU Count. To analyze the impact of CPU count on TLB shootdown overhead, we run our test program with core counts varying between 1 and 20 on a single socket on our test system. Fig. 1a shows the results.

For 1 core, no TLB shootdown IPIs are sent as intuitively expected. From 2 to 16 cores, the number of IPIs grows linearly. This illuatrates the limitations of system-level solutions: the OS has no knowledge of TLB contents and must send IPIs to all the cores using the virtual address space to guarantee correctness. Above 16 cores, the number of TLB shootdown IPIs no longer increases, as the program can not use more than 16 CPUs. The execution time and number of cycles in fig. 1a reflect the increase in IPIs, indicating a linear relationship between CPUs used by the program and the cost of TLB shootdowns.

3.2.2. NUMA. We assess the impact of NUMA on TLB shootdown overhead by executing our test program pinned to 12 cores, spread over 1 to 4 sockets on our test system. Fig. 1b shows that indeed both execution time and cycles rise with the number of sockets. Combining the results from fig. 1a and fig. 1b enables us to estimate just how much more

1. https://linux.die.net/man/1/taskset
2. https://linux.die.net/man/1/perf
3. https://github.com/StijnSchildermans/tlb_shootdown_mitigation.git



(a) `CPU count` (b) `Socket count` (c) `Virtualization`

Figure 1. Impact of several system properties on TLB shootdown overhead.

expensive remote IPIs are. Given that the total number of cycles required to execute the workload is the sum of the cycles spent on IPIs to the same socket, IPIs to a different socket, and a constant representing the remainder of the code, we can derive the following:

$cycles = \frac{a \times IPIs}{sockets} + (n \times a) \times IPIs(1 - \frac{1}{sockets}) + C$

$a = cycles(IPI_{local})$

$n = \frac{cycles(IPI_{remote})}{cycles(IPI_{local})}$

$C = cycles(remaining\ code)$

When we substitute C for the amount of cycles spent on the one CPU variant from fig. 1a, and $IPIs$ for the number of IPIs sent by the 12-CPU variant of the test program, we can determine a and n by curve fitting the above formula to the results from fig. 1b. We find a near-perfect fit for $a = 3200$ and $n = 3$. This indicates that IPIs sent to a remote socket are approximately 3 times as expensive as those sent to CPUs on the local socket. The solid lines on figure 1b represent the determined curve (adjusted accordingly for execution time).

3.2.3. Virtualization. As stated in §2, virtualization can increase the cost of TLB shootdowns in 3 ways: VM exits, system architecture abstraction and TLB shootdown preemption. To test the impact of these issues, we ran our test program on a VM with 16 vCPUs, and compare the cycles and time used to native execution. We experiment with APICv turned on and off, with multiple corunners (other VMs competing for the same CPUs) to induce TLB shootdown preemption, and with the vCPUs spread over multiple sockets. Fig. 1c shows the results.

The results from fig. 1c indicate that even basic use of virtualization decreases application performance by a factor of roughly 3. When additionally TLB shootdown preemption and NUMA abstraction occur, an order of magnitude of performance degradation is observed relative to native execution. Keep in mind that for each of the scenarios in fig. 1c, the system is identical from the perspective of the guest! Truly concerning is the fact that in the current era of cloud computing virtualized and overcommitted NUMA systems, represented by the worst-case scenario in fig. 1c, are becoming the standard application deployment platform. This clearly illustrates that now more than ever is the time to tackle the long-standing issue of TLB shootdown overhead.

4. Memory Management & TLB Shootdowns

As noted in §1, memory management may induce high TLB shootdown overhead. Throughout the years many different memory allocators have been developed. While the internals of these allocators may vary wildly, concerning TLB shootdowns we are only interested in how they interact with the system. In this regard, only a handful of principles are commonly applied. We discuss each of these below.

4.1. Hysteresis-Based Arenas

Early memory allocators were poorly scalable since they used a global lock, serializing any heap modification. To alleviate thread contention, the heap was divided in autonomous arenas, each protected by their own lock. Arenas are strictly isolated. As such, OS-interaction is fine-grained and happens on a per-arena basis. To avoid excessive system interaction when the memory footprint of an arena changes hysteresis is employed in the form of padding when the heap is expanded and a trim threshold which must be exceeded before it is shrunk. Many memory allocators are based on this concept, most notably glibc's ptmalloc2 [13].

While resizing arenas aggressively on a per-thread basis is efficient, many resizing operations induce increasingly costly TLB shootdowns, as described in §3. Moreover, the global memory efficiency gained by resizing an arena may be minor as individual arenas may only hold a fraction of the application memory and multiple arenas expanding and shrinking simultaneously may balance each other out. There may thus be an imbalance between the rate at which the (relative) memory footprints of individual arenas and the application change, suggesting that aggressively resizing arenas may not be worth the cost from an application-wide perspective. We call this the *arena imbalance issue*. Listing 1 shows a workload suffering from this issue if executed by multiple threads in parallel using ptmalloc2.

```
void* work(void* arg){
void* m[1000];
for (int i = 0; i < 1000; i++){
 for(int j=0;j<1000;j++)m[j]=malloc(130048);
 for(int j=0;j<1000;j++)free(m[999-j]);
}}
```

Listing 1. Minimal program causing many TLB shootdowns in ptmalloc2.

The workload from listing 1 allocates 1000 chunks of 127kB of memory. Then, all the memory is deallocated in reverse order. This process is contained in a loop. When 16 threads execute this workload in parallel on our test system described in §3.1, this results in 230 million TLB shootdown IPIs. Interestingly, if the order of freeing the chunks is reversed in listing 1, TLB shootdowns and program runtime are reduced by resp. 99.8% and 87% since heaps can not be shrunk when the top chunk is in use. This indicates that the arena imbalance issue can be introduced by minor changes to source code and have a severe performance impact.

4.2. Decay-Based Purging

Some arena-based allocators do not use hysteresis to determine when memory should be returned to the system, but employ decay-based purging. Freed memory is gradually released to the OS after a set amount of real time has elapsed (typically seconds). While this largely mitigates the arena imbalance issue, a capacitive effect is introduced. For applications with a rapidly and strongly varying memory footprint throughout their execution, decay-based purging is significantly less efficient than hysteresis-based trimming. The most popular decay-based memory allocator is FreeBSD's jemalloc [14].

4.3. Size-Class-Based Memory Management

Some allocators do not use arenas at all. Instead, they use per-thread caches which consist of a series of bins containing a list of chunks of fixed size classes, which are replenished in batches from a central heap. Memory is very coarsely returned to the system (based on hysteresis) since each size class must retain some padding. While this concept exhibits excellent performance, it is known to be especially susceptible to fragmentation since freed chunks can only be recycled by allocations of the same size class, leading to low memory efficiency [15]. Many high-performance allocators use this concept, such as tcmalloc [16] and memcached [17].

4.4. Garbage Collection

Today, most allocators employ garbage collection. In contrast to the allocation mechanisms described above, memory management is entirely performed by an algorithm, without programmer intervention. Because this algorithm is very expensive, it is deferred as long as possible. For example, in Java, a large amount of memory is reserved when the program starts and the garbage collector is only run when (a generation of) the heap is full [18].

While deferring garbage collection may minimize performance overhead -including TLB shootdowns-, memory efficiency suffers greatly since heap sizes are altered only sporadically and coarsely. Many studies have found that garbage collection has a detrimental impact on memory efficiency [19]–[21]. Thus, when efficiency is a concern, allocators employing garbage collection are not an option. Examples of memory allocators employing garbage collection include those used by Java, .NET, Python, etc.

5. Rethinking Memory Allocator Scalability

§4 has shown that no memory allocator combines excellent memory efficiency with negligible TLB shootdown overhead. This thus appears to be a fundamental design trade-off. However, when studying relevant literature (see §8), we found that this trade-off is never explicitly considered. Rather, the main trade-off is relieving thread contention (favored by high-performance allocators) versus maximizing memory efficiency (favored by high-efficiency allocators). The low TLB shootdown overhead many allocators exhibit is thus a side effect of other design decisions rather than a design goal. Notwithstanding, the results from §3 demand design priorities to be reconsidered.

An explicit focus on the trade-off between memory efficiency and TLB shootdown overhead promises to yield an allocator that balances these traits better than any other. This can even be achieved without sacrificing other important characteristics by starting from an existing concept and refining it. To this end, we devised the concept of *global hysteresis*, which is based on that of hysteresis-based arenas (see §4.1). We describe this concept below.

5.1. Global Hysteresis

Balancing efficiency and scalability for hysteresis-based arenas equates to eliminating the arena imbalance issue described in §4.1 with a minimal impact on memory efficiency. To come up with a viable concept, we asked ourselves the following fundamental question:

Does the change to the memory footprint of the application justify the performance overhead of resizing the arena?

To answer this question, we must know the benefits of a pending arena resizing operation regarding an application's memory usage as well as the cost of a TLB shootdown. Knowing these, we may allow for low memory efficiency in individual arenas when the global impact thereof is minor relative to the cost of resizing the arena. This mandates a global notion of the application state, in contrast to classic hysteresis which only considers the arena to be resized. We must thus partially break the strict isolation between arenas, allowing basic usage statistics to be exchanged in order to determine appropriate hysteresis thresholds. We thus introduce *global hysteresis*.

The TLB shootdown cost of resizing an arena can be estimated from the number of CPUs the application is using, as described in §3.2.1. The global memory usage implications thereof can be known by iterating over all arenas and calculating their cumulative memory usage. From this, a suitable top padding and trim threshold can be determined. These values may be much larger than those we would have chosen if only the arena to be resized were taken into account. Arenas are only trimmed if the total trimmable space of the application exceeds the determined threshold. We thus mitigate the arena imbalance issue at the cost of some memory efficiency and thread contention. If the hysteresis thresholds are chosen intelligently, we are confident the benefits of this approach outweigh the cost.

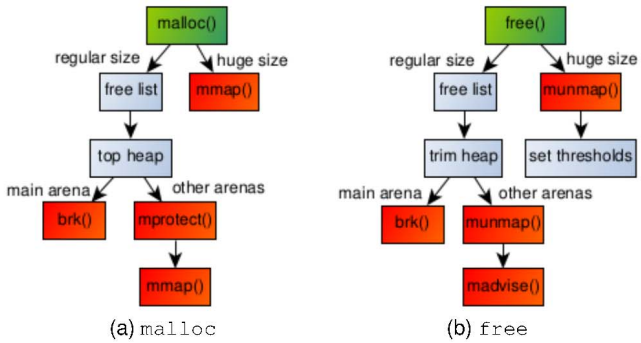


Figure 2. Simplified schematic overview of ptmalloc2.

6. Implementing Global Hysteresis

We implemented global hysteresis to validate its potential. As stated in §5, we start from an existing allocator using hysteresis-based arenas. We chose ptmalloc2 for this purpose as it is widely used, open source and well documented. We implemented our concept as an open-source library named *ptlbmalloc2*[4] on top of ptmalloc2 to allow existing projects to easily adopt our solution. Below we first describe relevant implementation details of ptmalloc2, after which we elaborate on how ptlbmalloc2 improves on them.

6.1. Ptmalloc2

Fig. 2 provides a simplified schematic overview of the workings of the `malloc` and `free` routines of ptmalloc2, which are used to resp. allocate and free memory [13].

Ptmalloc2 usually allocates a dedicated arena for each thread. Arenas may consist of multiple contiguous memory regions (heaps). Large `malloc` calls are served directly using the `mmap` system call. For smaller chunks, a variety of bins is traversed in search of a suitable previously freed block. If this search is fruitless, the block is allocated from the top of the arena; if needed after first expanding it. For the main arena expansion is achieved through the `brk` system call, and extra padding is added for future allocations. For other arenas, `mprotect` is used as long as the top heap can be expanded. If not, a new heap is added using `mmap`.

When a large chunk is freed, it is returned to the system using `munmap`, and the hysteresis thresholds are updated based on the size of the freed chunk. Smaller chunks are added to one of the bins. Next, the arena is trimmed if its free top space exceeds the trim threshold, leaving a small amount of padding for later allocations.

Glibc uses several data structures to track the state of chunks, heaps and arenas. These can be accessed based on the pointer returned by `malloc`. Based on this meta data, the state of the entire memory space can be queried.

`Brk`, `munmap`, `mprotect`, and `madvise` all induce TLB shootdowns (see §2). This causes the overhead associated with the arena imbalance issue described in §4.1.

4. https://github.com/StijnSchildermans/tlb_shootdown_mitigation.git

min	max	threshold
0 B	500 kB	100 kB
500 kB	1 MB	50%
1 MB	1 GB	10% + 400 kB
1 GB	∞	100 MB

TABLE 1. BASE THRESHOLDS USED IN PTLBMALLOC2

6.2. Ptlbmalloc2

§6.1, identifies three mechanisms principally responsible for the arena imbalance issue and ensuing TLB shootdowns in ptmalloc2: threshold calculation, claiming memory and trimming arenas. Below we discuss how we alter each of these by implementing global hysteresis.

6.2.1. Threshold Calculation. In ptmalloc2, the padding and trimming tresholds apply to individual arenas, and are set based on the size of individual chunks. In ptlbmalloc2 on the other hand, the thresholds apply to the entire application and are based on the total memory usage. To achieve this, we maintain a global array containing pointers to all of ptmalloc2's internal arena data structures. On each `malloc` call, we add the arena of the newly allocated chunk to this data structure, if it was not already present. On each free call, we call ptmalloc2's `free` routine and check if this has changed the size of the arena the chunk belonged to significantly. If so, we calculate the cumulative memory usage of all arenas by iterating over the list of arena pointers, locking each arena and querying its metadata. Based on this value, we heuristically determine a base threshold value (see table 1). To take the current application CPU usage into account as prescribed in §5, we program a periodic interrupt at a rate of 1Hz which calls the `times` system call, yielding the CPU time used by the program. Based on this, we can determine the average number of CPUs used in the past second. We then multiply the threshold by $1 + \frac{CPUs}{100}$. This yields the trimming threshold used by ptlbmalloc2. We set the top padding threshold equal to 25% of this value.

6.2.2. Claiming Memory. In contrast to ptmalloc2, ptlbmalloc2 preemptively applies top padding to all arenas. The amount of padding is determined by dividing the global padding threshold from §6.2.1 by the number of arenas. After every `malloc` call, we determine if the usable top space of the arena is at least 25% of this value. If not and the heap can still be expanded, we call `mprotect` to set the desired top padding. Ptmalloc2's internal data structures are updated to be consistent with our changes.

6.2.3. Trimming Arenas. On the first call to `malloc`, we disable ptmalloc2's heap trimming using `mallopt`. Whenever we iterate over arenas to determine hysteresis thresholds (see §6.2.1), we also calculate the cumulative free top space. If this value exceeds the trim threshold, we trim all arenas who's top space exceeds twice the per-arena top padding from §6.2.2, retaining only said top padding. The main arena is trimmed using the built-in glibc function `malloc_trim`. For other arenas, `madvise` is used.

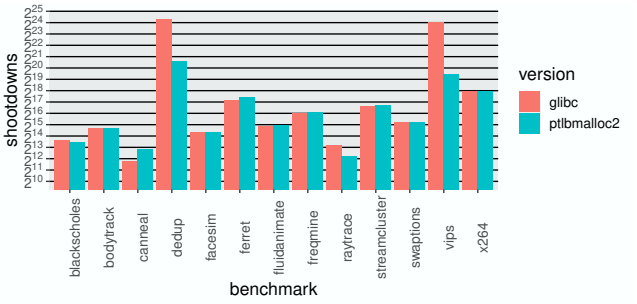


Figure 3. Comparison of TLB shootdowns for the Parsec benchmarks using ptmalloc2 and ptlbmalloc2; run natively with 16 CPUs on 1 socket.

7. Evaluation

We evaluate ptlbmalloc2 by comparing it to ptmalloc2 using a variety of system configurations and multithreaded workloads. All experiments are performed on the system described in §3.1. For the workloads we choose the *PARSEC 3.0* benchmark suite, which is widely used and covers a broad application domain [22]. A large body of existing work compares ptmalloc2 to other memory allocators, facilitating extrapolation of our results [15], [23].

7.1. Conceptual Effectiveness

We first evaluate to what extent the main goal of global hysteresis embodied in ptlbmalloc2 has been achieved: eliminating the arena imbalance issue. To that end, fig. 3 compares the number of TLB shootdowns for all PARSEC benchmarks using resp. ptmalloc2 and ptlbmalloc2, run natively using 16 CPUs on one socket on a logarithmic scale.

Fig. 3 shows that for most benchmarks, the number of TLB shootdowns is low using ptmalloc2. This is to be expected, since the arena imbalance issue is only induced by specific allocation patterns. However, *dedup* and *vips* do exhibit many TLB shootdowns. These are highly likely to be induced by the arena imbalance issue since page migrations and I/O can be eliminated as causes due to the nature of the workloads and the system. Ptlbmalloc2 eliminates almost all TLB shootdowns for these benchmarks without significantly affecting others. This indicates that global hysteresis is a viable concept to eliminate the arena imbalance issue.

7.2. Side Effects

While §7.1 indicates that ptlbmalloc2 achieves its main goal, it may have undesirable side effects such as increased resource usage. To gain insight into this, we analyzed three principal memory allocator performance metrics: execution time, memory efficiency and CPU cycles. We asses ptlbmalloc2 relative to ptmalloc2 regarding these metrics for all PARSEC benchmarks with CPU counts varying from 4 to 64, spread over 1 to 4 sockets. Fig. 4 shows the results at the extremes of the tested system configurations. Other configurations reliably yield results between these extremes.

Globally, the results in fig. 4 align with expectations. In fig. 4b, both *dedup* and *vips* show a noticeable speedup. In fig. 4a however *vips* consumes slightly more cycles and time, while memory efficiency is 5% better compared to ptmalloc2. This is possible when a benchmark allocates many large chunks. In ptmalloc2, the trim threshold keeps increasing as the mmap threshold increases. Ptlbmalloc2 on the other hand bases its thresholds on the application state and may shrink them accordingly. It can thus be more memory efficient than glibc at a minor cost in performance.

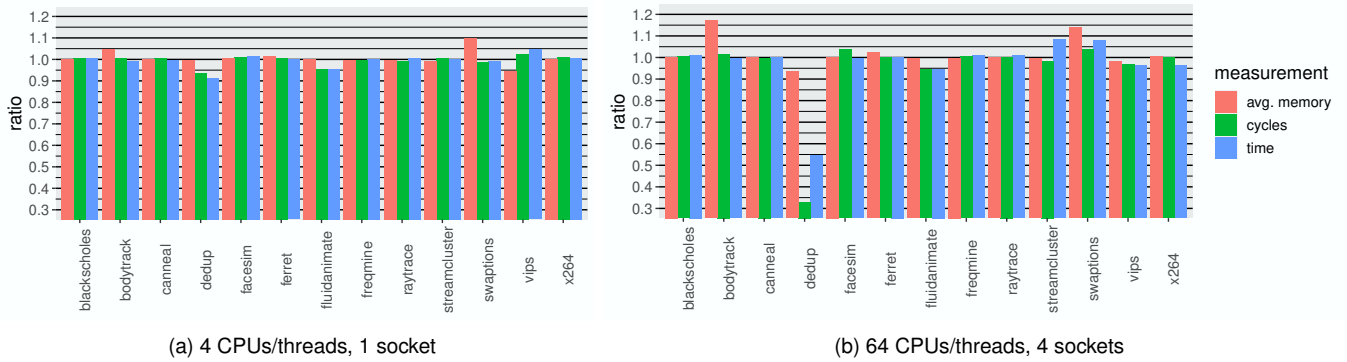
Curiously, *fluidanimate* consistently shows a performance improvement of $\pm$ 5% despite not suffering from the arena imbalance issue, as indicated by fig. 3. Closer analysis reveals that that this is not a direct consequence of the design considerations of ptlbmalloc2, as the number of system calls performed by this benchmark is identical for ptlbmalloc2 and ptmalloc2. Rather, improved cache performance causes this result. Because cache behavior is very complicated, not a focus of ptlbmalloc2's design and out of scope of the paper, we refrain from attributing this result to hypothetical superior design of ptlbmalloc2. For the same reasons we leave a deeper analysis of this finding for future work.

Other benchmarks show performance very close to ptmalloc2. None show a consistent significant performance degradation across system configurations. Moreover, after carefully analyzing the benchmarks exhibiting a mild slowdown in ptlbmalloc2, we found that the main cause is increased thread contention for arenas. This is partly by design as explained in §6, and partly because we did not integrate our code into ptmalloc2 itself, requiring us to contend with ptmalloc2 code for arena locks. By integrating ptlbmalloc2 directly in glibc, we could eliminate the majority of this contention at the cost of reduced flexibility.

Memory efficiency is overall much better than we expected. Only *bodytrack* and *swaptions* show notably increased memory usage, which never exceeds 15%. After analyzing the memory profile we found that these benchmarks consume very little memory (30MB for *bodytrack* and 4MB for *swaptions*). These results are thus very acceptable.

We finalize our analysis of side effects by studying ptlbmalloc2's performance in a virtualized scenario. Fig. 5 shows the results in terms of cycles. We omit the other metrics as execution time is in line with cycles in fig. 5 and memory efficiency is in line with that shown in fig. 4.

As expected, fig. 5 shows that performance improvements are greater in virtualized scenarios for the benchmarks suffering from the arena imbalance issue. All other benchmarks, with the exception of swaptions, perform nearly identical to or better than ptmalloc2. Moreover, several outliers can be seen where ptlbmalloc2 performs much better than ptmalloc2 for specific configurations (e.g. *canneal*, 64 CPUs and *streamcluster*, 4 CPUs, 4 sockets). These benchmarks induce many TLB shootdowns only in specific scenarios. This shows that ptmalloc2 is also susceptible to platform specifics, thus providing strong evidence that occasional limited relative performance variations between ptlbmalloc2 and ptmalloc2 are bidirectional.



(a) 4 CPUs/threads, 1 socket

(b) 64 CPUs/threads, 4 sockets

Figure 4. Average memory usage, execution time and cycles for the PARSEC benchmarks using ptlbmalloc2 relative to ptmalloc2 in various scenarios.

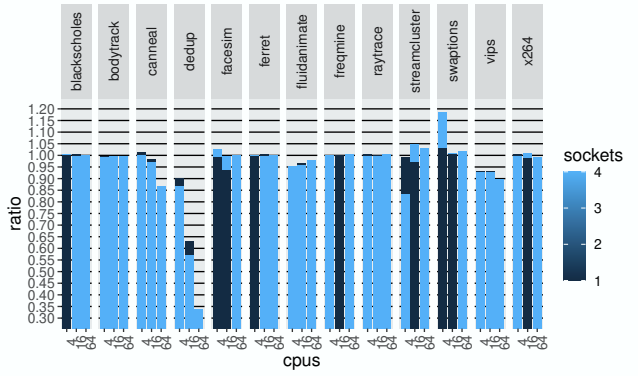


Figure 5. Cycles used by ptlbmalloc2 relative to glibc for the Parsec workloads in a virtualized environment.

Env.	CPUs	Sockets	Speedup
native	4	1	0%
native	4	4	-1%
native	16	1	1%
native	16	4	2%
native	64	4	5%
virt	4	1	1%
virt	4	4	2%
virt	16	1	4%
virt	16	4	4%
virt	64	4	7%

TABLE 2. AVERAGE PERFORMANCE IMPROVEMENT OF PTLBMALLOC2 ACCROSS ALL PARSEC BENCHMARKS IN ALL TESTED SCENARIOS.

7.3. Performance

Distilling the results from §7.2 we find that ptlbmalloc2 greatly improves performance for benchmarks suffering from the arena imbalance issue in ptmalloc2. Most other benchmarks behave nearly identical to ptmalloc2, with minor exceptions in both directions. To gain a conclusive insight into the performance of ptlbmalloc2, we summarize the cycles used relative to ptmalloc2 in table 2 as an average of all benchmarks for all studied system configurations.

Table 2 shows that on average, ptlbmalloc2 almost always outperforms glibc. Performance improvement rises drastically with CPU count. In virtualized environments the impact is even greater. To our surprise, NUMA configuration has only a limited effect. The average of all results in table 2 is 3%. We thus conclude that ptlbmalloc2 boasts modest performance improvements on average, with some benchmarks benefiting greatly, while most other benchmarks are at most mildly affected. Ptlbmalloc2 thus succeeds in its goal of eliminating the arena imbalance issue, while having a minimal impact on other aspects of the memory allocator.

8. Related Work

TLB performance has been studied extensively in literature. However, most studies focus on increasing TLB hit rate or reducing TLB miss latency without directly addressing TLB shootdowns [24]. The ones that do address the latter explicitly provide solutions at hardware [1], [11], [25] or system [2], [6] level. This poses real challenges to their applicability since at such at a low level many side effects must be considered and adoption may take many years.

A few proposals argue for a radically different OS design to tackle a wide range of problems arising from the current trend of ever-growing core counts in shared-memory systems [26], [27]. Such OS designs view every CPU as a discrete entity. Each CPU runs its own microkernel and communication between CPUs is explicit. This reduces or even eliminates the need for OS-managed TLB consistency, among many other benefits. Although experimental implementations of such systems exist, it is unlikely that any will be widely adopted in the foreseeable future.

A limited body of work studies the impact of virtualization on TLB performance [4], [9], [10]. Most of these studies however fail to provide a generally applicable and readily adoptable solution. The few studies that do propose a tenable solution are highly specific in scope and focus on the system and hardware level, suffering the same drawbacks as mentioned above [3].

At application level, many memory allocators with a focus on scalability and efficiency have been developed. Some of these have had a significant impact on mainstream systems [14], [16]. However, in §4 we have shown that

no memory allocator combines excellent memory efficiency and (TLB) scalability because TLB shootdown overhead has never been an explicit design consideration. While many allocators exhibit low TLB shootdown overhead, this is a side effect of other design decisions trading memory efficiency for performance in a broader sense. Research efforts to improve these existing allocators are plentiful and ongoing, e.g. by ameliorating synchronization mechanisms [28] or data locality [29]. Such work is orthogonal to ours.

9. Conclusion & Future Work

Due to several evolutions in the nature of contemporary computing platforms TLB shootdown cost is steadily rising. Since for multithreaded applications many of these TLB shootdowns are caused by memory management at application level, optimizing memory allocators is a promising method to address this issue. Existing allocators either exhibit poor TLB performance due to the arena imbalance issue or poor memory efficiency due to a focus on performance aspects distinct from TLB consistency. By explicitly focussing on the trade-off between (TLB) scalability and memory efficiency a memory allocator design concept and implementation exhibiting excellent performance in both these metrics with minimal side effects have been introduced in this work: resp. global hysteresis and ptlbmalloc2.

While global hysteresis achieves its objectives, it is tightly bound to a specific legacy allocator design concept. We believe the core issue is much broader: the trade-off between memory efficiency and performance. Given the recent increase in TLB shootdown cost, memory allocators in general must reconsider how they interpret the metric 'performance'. We aim to address this abstract issue in future work and plan to devise a general conceptual solution, of which global hysteresis is one possible incarnation.

Acknowledgments

We thank the anonymous reviewers for their constructive comments. This work is funded in part by the US National Science Foundation grant CCF 1617749, the Flemish FWO grant V433819N and KU Leuven JUMO grant 19005.

References

[1] B. F. Romanescu, A. R. Lebeck, D. J. Sorin, and A. Bracy, "UNified instruction/translation/data (UNITD) coherence: One protocol to rule them all," in *HPCA 2010*, pp. 1–12.

[2] M. K. Kumar, S. Maass, S. Kashyap, J. Veselỳ, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna, "Latr: Lazy translation coherence," in *ASPLOS 2018*, pp. 651–664.

[3] J. Ouyang, J. R. Lange, and H. Zheng, "Shoot4U: Using VMM assists to optimize TLB operations on preempted vCPUs," *VEE 2016*.

[4] S. Schildermans and K. Aerts, "Towards high-level software approaches to reduce virtualization overhead for parallel applications," in *CloudCom*, 2018, pp. 193–197.

[5] N. Amit, A. Tai, and M. Wei, "Don't shoot down TLB shootdowns!" in *EuroSys 2020*.

[6] N. Amit, "Optimizing the TLB shootdown algorithm with page access tracking," in *USENIX ATC 2017*, pp. 27–39.

[7] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill, "Translation lookaside buffer consistency: a software approach," *ACM SIGARCH Computer Architecture News*, vol. 17, no. 2, pp. 113–122, 1989.

[8] I. Corporation. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf

[9] X. Ding and J. Shan, "Diagnosing virtualization overhead for multithreaded computation on multicore platforms," in *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on*. IEEE, 2015, pp. 226–233.

[10] R. McDougall and J. Anderson, "Virtualization performance: Perspectives and challenges ahead," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 4, pp. 40–56, Dec. 2010.

[11] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "Didi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory," in *PACT 2011*, pp. 340–349.

[12] Torvalds, "torvalds/linux," Jun 2019. [Online]. Available: https://github.com/torvalds/linux

[13] Emeryberger, "emeryberger/malloc-implementations," Jul 2012. [Online]. Available: https://github.com/emeryberger/Malloc-Implementations/tree/master/allocators/ptmalloc/ptmalloc2

[14] J. Evans, "A scalable concurrent malloc (3) implementation for freebsd," in *Proc. of the bsdcan conference, ottawa, canada*, 2006.

[15] D. Rentas, "Evaluate the fragmentation effect of different heap allocation algorithms in linux," 2015.

[16] S. Ghemawat and P. Menage, "Tcmalloc: Thread-caching malloc," 2009.

[17] A. Wiggins and J. Langston, "Enhancing the scalability of memcached," *Intel document, unpublished, http://software. intel. com/en-us/articles/enhancing-the-scalability-of-memcached*, 2012.

[18] Oracle, "Understanding memory management," Jan 2010. [Online]. Available: https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html

[19] S. Sangappa, K. Palaniappan, and R. Tollerton, "Benchmarking java against c/c++ for interactive scientific visualization," in *JGI 2002*, pp. 236–236.

[20] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23–29, 2000.

[21] P. Kulkarni, H. Kailash, V. Shankar, S. Nagarajan, and D. Goutham, "Programming languages: A comparative study," *Information Security Research Lab, NITK, Surathkal*, 2008.

[22] C. Bienia and K. Li, "PARSEC 2.0: A new benchmark suite for chip-multiprocessors," in *MoBS 2009*, June.

[23] R. Liu and H. Chen, "Ssmalloc: a low-latency, locality-conscious memory allocator with stable performance scalability," in *APSys 2012*, pp. 1–6.

[24] S. Mittal, "A survey of techniques for architecting TLBs," *Concurrency and computation: practice and experience*, vol. 29, no. 10, 2017.

[25] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level TLBs for chip multiprocessors," in *HPCA 2011*, pp. 62–63.

[26] E. Zurich, "The barrelfish operating system," Oct 2018. [Online]. Available: http://www.barrelfish.org/index.html

[27] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. T. Morris, A. Pesterev, L. Stein, M. Wu, Y.-h. Dai *et al.*, "Corey: An operating system for many cores." in *OSDI 2008*, pp. 43–57.

[28] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "Radixvm: Scalable address spaces for multithreaded applications," in *EuroSys 2013*, pp. 211–224.

[29] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos, "Scalable locality-conscious multithreaded memory allocation," in *ISMM 2006*, pp. 84–94.