A Scalable Pthreads-Compatible Thread Model for VM-Intensive Programs

Yu Zhang(✉) and Jiankang Chen

University of Science and Technology of China, Hefei, China
yuzhang@ustc.edu.cn

Abstract. With the widespread adoption of multicore chips, many multithreaded applications based on the shared address space have been developed. Widely-used operating systems, such as Linux, use a per-process lock to synchronize page faults and memory mapping operations (e.g., `mmap` and `munmap`) on the shared address space between threads, restricting the scalability and performance of the applications. We propose a novel Pthreads-compatible multithreaded model, PATHREADS, which provides isolated address spaces between threads to avoid contention on address space, and meanwhile preserves the shared variable semantics. We prototype PATHREADS on Linux by using a proposed character device driver and a proposed shared heap allocator *IAmalloc*. Pthreads applications can run with PATHREADS without any modifications. Experimental results show that PATHREADS runs 2.17×, 3.19× faster for workloads `hist`, `dedup` on 32 CPU cores, and 8.15× faster for workload `lr` on 16 cores than Pthreads. Moreover, by using Linux Perf, we further analyze critical bottlenecks that limit the scalability of workloads programmed by Pthreads. This paper also reviews the performance impact of the latest Linux 4.10 kernel optimization on PATHREADS and Pthreads, and results show that PATHREADS still has advantage for `dedup` and `lr`.

1 Introduction

With the widespread adoption of multicore chips, many multithreaded applications based on the shared address space have been developed. Such shared address space, however, requires the kernel virtual memory (VM) subsystem to synchronize concurrent address space operations (i.e., page faults or `mmap`, `munmap` and `mprotect` system calls) launched by either application code or the called libraries, e.g., `libc`. Widely-used operating systems such as Linux often use a single read-write lock per process to serialize address space operations, limiting the scalability of multithreaded VM-intensive applications [9]. For example, on a 32-core Linux system mentioned in Sect. 5, `histogram` (`hist`) from Phoenix [20] consumes up to 40% and 50% of the execution time in the kernel per-process lock on 16 and 32 cores[1], respectively.

[1] The number of threads (workers) equals to the number of CPU cores enabled.

J. Vaidya and J. Li (Eds.): ICA3PP 2018, LNCS 11337, pp. 121–137, 2018.
https://doi.org/10.1007/978-3-030-05063-4_11

Some research efforts try to improve scalability of multithreaded software. Clements *et al.* propose a scalable address space by using RCU balanced trees [9]. Kleen *et al.* think that processes scale better than threads since a process does not share the address space with others [15]. Psearchy from MOSBENCH [7] was modified to use processes instead of threads. However, due to shared variables between threads, it is quite difficult to migrate a multithreaded workload from the shared address space to thread-private address space. In addition, super page is also used to reduce the number of page faults so as to ease contentions [9], but cannot benefit from many small address regions.

Unlike the above work, we propose a new scalable thread model, PATHREADS (Private Address space based threads), which adopts isolated address space for each thread to avoid contentions on address space operations, but keeps shared variable semantics on the heap and globals. Like Pthreads, each thread in PATHREADS has its own stack, and can synchronize with other threads via locks, condition variables or barriers using the same Pthreads API. Thus, Pthreads applications can be built and run with PATHREADS without any code modifications.

To reduce contentions on the shared heap, we design a novel shared heap allocator, called *IAmalloc* (Isolated Address space based malloc). *IAmalloc* provides each thread an isolated address range for allocation, but allows all threads to read or write the whole heap. The isolation of the allocation area can avoid synchronous operations at the time of allocation, and can further provide good spatial locality when allocated heap objects are only visited by the thread that created them.

We prototype PATHREADS on Linux by emulating each thread using a single-threaded Linux process, where private copy-on-write (COW) memory mappings of a process can further be changed to meet the mapping requirement of PATHREADS via a proposed character device driver CDEV. Thus, users only need to install the device module in their own kernel without any kernel modifications. We further prototype *IAmalloc* by adopting Doug Lea's *dlmalloc* [16] with minor modification, and implement Pthreads synchronization API for PATHREADS using algorithms similar to Pthreads.

The main contributions of this paper are as follows:

- We analyze the reason why the VM-intensive multithreaded program has the performance bottleneck in the shared address space (see Sect. 2).
- We propose a scalable PATHREADS model and a shared *IAmalloc* allocator, providing thread-private address space but keeping shared variable semantics (see Sect. 3).
- We propose a solution to implement PATHREADS on Linux using well-designed "thread emulated by process" policy and character device driver (see Sect. 4).
- We prototype PATHREADS and demonstrate its effectiveness on a 32-core machine with Linux 3.2.0 and 4.10 kernels (see Sect. 5).

On Linux 3.2.0 kernel, it achieves up to 3.19×, 2.17× faster than Pthreads for `dedup`, `hist` on 32 CPU cores, respectively. PATHREADS with *IAmalloc* can also reduce false sharing and improve the performance of non-VM-intensive programs, e.g., achieving 8.15× faster than Pthreads for `linear_regression` (`lr`) on 16 cores. Although the latest Linux kernel has significant improvement on the kernel locking mechanism, PATHREADS still has advantage for some Pthreads programs, e.g., achieving up to 2.99× faster than Pthreads for `dedup` on 32 cores, and 7.45× faster for `lr` on 16 cores.

2 Background and Motivation

In this section, we first introduce the VM management in Linux Kernel, then describe VM-intensive programs and their performance bottlenecks.

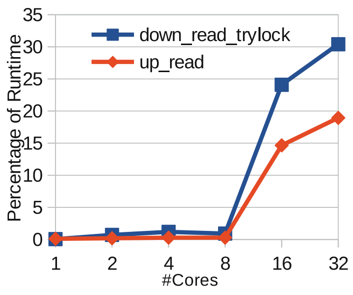
VM Management and Contention in Linux Kernel. In most widely used operating systems such as Linux, an address space shared among threads principally consists of a set of memory mapping regions and a page table tree. Linux uses struct *mm_struct* to manage the whole address space among threads, and struct *vm_area_struct* (VMA) to describe a memory mapping region including its start address, end address and flags to determine access rights and behaviors.

The *mm_struct* includes a red-black tree to store VMAs in order to enable the OS kernel to quickly find a region containing a particular virtual address. Linux provides system calls for memory mapping operations: `mmap`, `munmap` and `mprotect`. The `mmap` creates a memory mapping region and adds it to the tree, the `munmap` removes a region from the tree, and the `mprotect` updates the permission of a region by modifying the relative VMA. In addition, when a soft page fault is triggered, the handler looks up the tree and checks whether the faulting virtual address is mapped.

To ensure correct behavior when threads perform memory mapping operations and page faults concurrently, Linux uses a per-process read-write semaphore *mmap_sem* to serialize those operations launched by different threads. Therefore, a thread need acquire the semaphore in *write* mode before executing a memory mapping operation in order to prohibit other threads performing address space operations. And the page fault handler need acquire the semaphore in *read* mode and can proceed with other page faults in parallel. Consequently, contentions on *mmap_sem* become intense when running an application with a large number of threads, leading to speedup degradation.

Performance Bottleneck of VM-Intensive Programs. The VM-intensive program is a very important class of multithreaded programs, and its main feature is to contain a large number of address space operations. For example, programs frequently trigger page faults, or programs containing frequent or bulk heap operations would frequently call memory mapping operations. A page fault causes to acquire *mmap_sem* in read mode, while a memory mapping operation need acquire *mmap_sem* in write mode. Performance bottlenecks of the VM-intensive program in read or write modes are different. We next analyze them using the evaluation methodology in Sect. 5.

Contention in Read Mode. In order to find out the main reason of the performance loss, we test the overhead of hotspot functions for the workload. We find that the first two hotspot functions of `hist` are `down_read_trylock()` (trylock for reading) and `up_read()` (release a read lock) from Linux kernel. As shown in Fig. 1(a), overheads brought by the two functions increase heavily as CPU core count grows, and even reach up to 30% and 20% of total runtime on 32 cores, respectively. We then use Linux Perf [19] to find that these function calls are all derived from page faults. From the process of `do_page_fault()` shown in Fig. 1(b)), we know that with high CPU core count, the lock contention would be further aggravated by page faults.

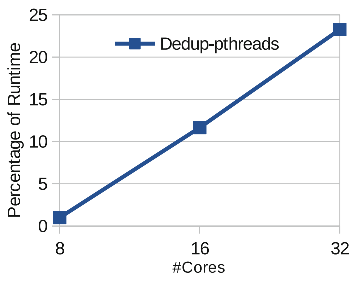


(a) `histogram`

```
PSEUDO-CODE: page fault
do_page_fault    () {
      ..........
       if(!down_read_trylock(mmap_sem)){
                 if(is bad exception)
                          return;
                 else
                 down_read(mmap_sem);
       }
       vma=find_vma(...)
       fault=handle_mm_fault(...);
       up_read(mmap_sem);
       ..........
}
```

(b) Process of page fault handling

Fig. 1. Overhead of contention in read mode and the process of page fault handling.



(a) `ticket_spin_lock`

```
|__ticket_spin_lock
|  |--rwsem_down_faild_common
|  |   --call_rwsem_down_read_failed
|  |      --down_read
|  |      --do_page_fault
|  |      --page_fault
|  |   --call_rwsem_down_write_failed
|  |      --down_write
|  |      --sys_mmap_pgoff
|  |      --sys_mmap
|  |      --system_call
|  |      --mmap64
```

(b) call-stack information

Fig. 2. Overhead of contention in write mode and the call stack information.

Contention in Write Mode. `dedup` from PARSEC [3] is a typical VM-intensive program. It suffers from serious lock contention, where the kernel `ticket_spin_lock()` occupies 24.7% of total execution time on 32 CPU cores, as shown in Fig. 2(a). Call-stack information in Fig. 2(b) generated by Linux Perf shows that `ticket_spin_lock()` is mainly invoked by the Linux kernel function `rwsem_down_failed_common()` (wait for a lock to be granted). There are two main sources of `rwsem_down_failed_common()`: one is caused by page fault handler if it fails to contend *mmap_sem* in read mode, and the other is caused by memory mapping system calls such as `mmap`, `mproctect` if they fail to contend *mmap_sem* in write mode. Thus we can draw a conclusion that the *mmap_sem* becomes a serious bottleneck for `dedup` using Pthreads.

3 The PAthreads Model

3.1 Design of the Thread Model

To tackle the above performance issues caused by the contention on per-process read-write semaphore, we propose a novel thread model, PAthreads in Fig. 3, which follows Pthreads programming interface but strives to achieve the following goals.

G1. Avoid contentions on the address space. We address the issue by confining threads in separate address spaces. Each thread has readonly (RO) text (code) segment and its own stack but occupies the same address segment (① in Fig. 3).

G2. Preserve the sharing semantics on heap objects and globals like Pthreads in order to support Pthreads programs without modification. We address the issue by designing mechanisms to change the VM mapping properties of the same address ranges in different address spaces, and to implement Pthreads synchronization API.

G3. Reduce contentions on the shared heap. We address the issue by letting each thread allocate heap objects in a disjoint subheap, but allow other threads accessing all subheaps (② in Fig. 3, where the bold line means allowing allocation).

Isolated Address Space. Each PAthreads thread has its own memory mapping structure (*mm_struct*) to manage its isolated address space without interference from other threads. Thus, there is no contention on the semaphore *mmap_sem* when performing address space operations even with high CPU core count.

However, the isolated address spaces among threads bring challenge in preserving the shared variable semantics among threads under good performance. In the next subsections, we introduce features of PAthreads to obtain the above G2 and G3.

3.2 Preserving Shared Variable Semantics

Threads running with Pthreads library share all memory except for the stack. Even if each thread has its own stack to maintain the control flow, a thread

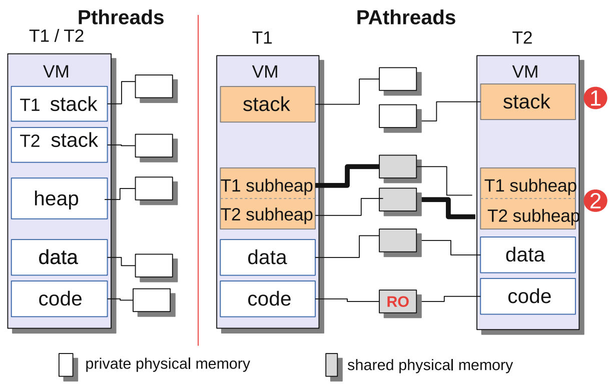


Fig. 3. Different thread models: Pthreads vs. PATHREADS

can read and write the stack content of other threads due to the shared address space. Changes to memory immediately become visible to all other threads. In general, threads often share two types of data with each other, i.e., globals in data segment and objects in heap segment, while the situation of stack sharing (especially write sharing) is relatively uncommon [17]. Improper use of stack write-sharing can cause significant security risks.

By contrast, since PATHREADS runs threads in separate address spaces, we have to consider how PATHREADS explicitly manage various shared data resources.

Stack. In view of the rareness and error-proneness of stack write-sharing, PATHREADS does not support stack write-sharing across threads, so any updates to stack variables are only locally visible. As shown in Fig. 3, Pthreads has to allocate a disjoint address range from the shared address space as each thread's stack. This causes that Pthreads cannot scale as the number of threads increases. But in the PATHREADS model, stacks of all threads occupy the same address range in their own address spaces, so as to effectively use the limited address space to support a large number of threads. When a child thread is created by its parent thread, it can directly inherit the stack status from the parent thread, accordingly supporting some meaningful and common stack read-sharing.

Globals and the Heap. In OSes such as Linux, a process invokes `fork()` to create a *child* process, which is an almost exact duplicate of the calling process, the *parent*. The two processes execute the same program text, but have separate copies of the stack, data and heap segments. The kernel employs COW mapping technique to avoid wasteful page copying but to allow each process modifying its private copies without affecting the other process. Multiple threads within a process share the process's address space and page tables, directly sharing globals and heap objects.

Nevertheless, in PATHREADS, in order to share globals and heap objects among threads, data and heap segments in each thread's separate address space should not be private COW mappings like Pthreads, but shared writable mappings. That is, the page-table entries for these segments in one thread refer to the same physical memory pages as the corresponding page-table entries in the other threads. As shown in the right side of Fig. 3, although threads T1 and T2 have their own address spaces, the data and heap segments belonging to T1 and T2 will map to the same shared physical pages. Thus, the shared variable semantics could be ensured in PAthreads.

Code. Similar like Pthreads, the text segment of each thread in PAthreads is marked as read-only, and is mapped to a common set of physical pages.

Synchronization Algorithms and Primitives. PAthreads should support programs that invoke Pthreads synchronization API, such as locks, condition variables and barriers. All these synchronization algorithms should be similar to Pthreads, but are built atop the separate address spaces of PAthreads, rather than the shared address space.

3.3 The Heap Allocator

In PAthreads, due to the thread-private address space, it is impossible to ensure heap sharing semantics by using legacy allocators. To address the problem, a new heap allocator, *IAmalloc*, is proposed to apply to the PAthreads. *IAmalloc* need ensure that a heap object allocated by one thread can be read or written correctly by the other threads, and it also need effectively support dynamic memory allocation, access and recycling.

To achieve the above requirements, *IAmalloc* first makes each thread occupy the same address range as its heap segment; then it divides the whole heap segment into several subheaps, and each subheap can only be used to respond allocation requests from a single thread; finally, *IAmalloc* ensures that all allocated heap objects in different subheaps can be read and written by each thread. Thus, *IAmalloc* avoids synchronization when allocating heap objects. Moreover, since heap objects allocated by a thread are in the same subheap, the spatial locality can be improved especially when the heap objects allocated by a thread would not be accessed by the other threads.

4 Implementing PAthreads on Linux

Prototyping PAthreads in a proof-of-concept OS or existing OS kernel is direct, but faces compatibility and practical issues. To prototype rapidly and make it practical, we implement PAthreads using a proposed character device driver on Linux.

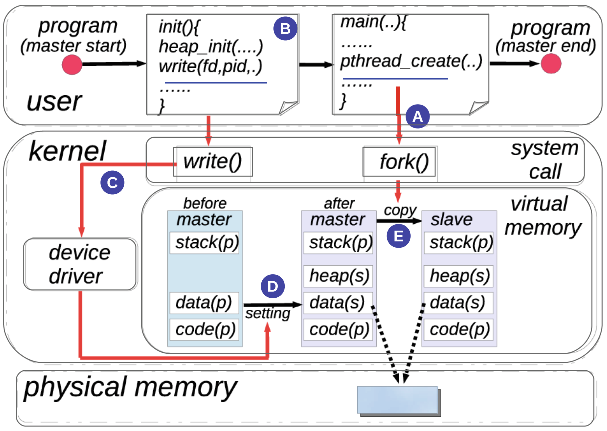


Fig. 4. Program execution flow, where *(p)* or *(s)* represent that the region is private or shared memory mapping, respectively.

4.1 Spawning a PATHREADS Thread

We emulate a PATHREADS thread using a single-threaded Linux process by invoking `fork()` in the implementation of `pthread_create()`. As mentioned in Sect. 3.2, a forked process has a private address space, which is an almost exact duplicate of its parent process via COW mappings. The forked process satisfies G1 (Sect. 3.1), but cannot support the shared variable semantics (G2) required by PATHREADS. Therefore, the **key** to implement PATHREADS is how to change the VM mapping attributes before any `pthread_create()` call in a Pthreads program.

To obtain shared globals in data segment and the shared heap, we first prohibit the main thread to execute COW on its data and heap segments, and set them shared. Subsequently, with the child threads created (Ⓐ in Fig. 4), they will have shared data and heap segments inherited from the main thread (Ⓔ in Fig. 4).

In order to prohibit COW, we need modify the properties recorded in VMAs of the master thread from private to shared. It is not practical to directly modify the source code of Linux kernel, because it will lead to rebuild the Linux kernel. To modify the kernel lightly, we develop a character device driver, CDEV, which follows the Linux device driver interface. Thus, users only need install the CDEV module into their own Linux kernel without any kernel modifications.

As depicted in Fig. 4, a special `init()` function (Ⓑ in Fig. 4) is added to perform the attribute modification of memory mappings. Such a function is decorated with GCC constructor attribute `__attribute__((constructor))`, so that it can be called automatically before entering `main()`. In the `init()`, we initialize the heap and then hijack the `write()` system call to access to the CDEV driver. Through `write()` (Ⓒ in Fig. 4), CDEV can get the process ID (PID) of the master thread, and find its *mm_struct*. CDEV then finds the VMA of the data and heap segments in the *mm_struct* and changes the mapping properties of the VMA from private to shared (Ⓓ in Fig. 4).

4.2 Heap Allocator *IAmalloc*

Due to the isolation among threads, traditional shared heap allocators such as *ptmalloc* [14] in `glibc` cannot apply to PAthreads. We then develop *IAmalloc* to preserve shared heap semantics among separate address spaces basing on Doug Lea's *dlmalloc* [16]. The **key point** is how to share heap between threads with good efficiency.

Heap Allocation Policy. Performing concurrent dynamic memory allocation and de-allocation in a single shared heap requires synchronization control each time, and would inevitably serialize all allocation and de-allocation operations, badly hurting the scalability [2]. The layout of the heap can have a significant impact on how fast the program is running [13]. To obtain efficiency, *IAmalloc* separates the entire heap into several subheaps, and each thread can get a subheap for thread-local allocation but can read or write any objects in the whole heap. Thus, it is unnecessary to synchronize dynamic allocations, and objects allocated by the same thread also have good spatial locality, accordingly improving the performance and scalability of programs.

In the implementation of *IAmalloc*, each subheap has corresponding metadata in the global structure shared among threads, and has a relative *mspace* structure managed by *dlmalloc*. *IAmalloc* directly reuses *mspace* memory management in *dlmalloc*. When a thread invokes `malloc()`, it first finds the *mspace* from its relative subheap without contention, then gets the desired memory block from the *mspace*. As shown in Fig. 5, threads T1 and T2 allocate space from their subheaps to store variables `x1` and `x2`, respectively. After that, both T1 and T2 can access `x1` or `x2`; and each thread can free a heap object allocated by another, e.g., T1 can free `x2`. In this situation, *IAmalloc* would search the global metadata of subheaps to find the target subheap including the to-be-freed memory address and then reclaim the memory block into the target.

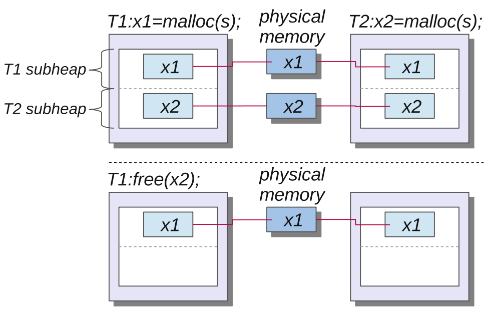


Fig. 5. T1 and T2 malloc x1 and x2, respectively; then T1 frees x2.

The Source of Chunks and Pad Allocation. In *dlmalloc*, if there is not enough memory in an mspace, the allocator would invoke `sbrk()` or `mmap()` to allocate a chunk from the address space. However, frequent accesses to the kernel must be a performance defect. To avoid this problem, we modify the algorithm of *dlmalloc*, that is, if a thread finds there is not enough virtual memory in the mspace, the thread would split a chunk with fixed size from an available pre-assigned subheap and then save it into the mspace.

False sharing occurs when processors access different data objects within the same cache line in parallel [5]. This false sharing can cause serious performance degradation. To reduce or even avoid false sharing, except for allocating in thread-local subheap, *IAmalloc* also pads the allocation, that is, makes the heap allocated addresses aligned.

4.3 Thread Management and Synchronization

Thread Management. When `pthread_create()` is called in application code, the PAthreads runtime would invoke `fork()` to create a child process (Ⓐ and Ⓔ in Fig. 4), assign a subheap to the child, and give the child a deterministic thread ID. Calling `pthread_self()` will return the assigned thread ID instead of the PID given by Linux. In order to implement `pthread_join()`, PAthreads maps the deterministic thread ID to PID returned by `fork()`, and invokes `waitpid()` to wait for the specified thread. The PAthreads runtime maintains both global and private meta-data for each thread, including PID, thread ID, the relative subheap, etc.

Synchronization. The implementation of *mutex* in PAthreads is similar to that in Pthreads, but works between multiple Linux processes rather than multiple threads within a Linux process. The PAthreads runtime lets a thread sleep instead of keeping busy while waiting for a mutex, and the algorithm of mutex is mentioned in [12].

A *condition variable* has a waiting queue and a mutex pointer in PAthreads. When a thread invokes `pthread_cond_wait()`, it need release the mutex and then sleep in the waiting queue. If the thread is waken up by a `pthread_cond_signal()` or `pthread_cond_broadcast()` call from another thread, this thread will contend the mutex. Algorithm about condition variables is discussed in [4].

In PAthreads, each *barrier* contains a condition variable and a mutex. If a thread is the last one arriving this barrier, it will wake up all threads sleeping in waiting queue. And the algorithm of barrier also comes from [12].

5 Evaluation

We first introduce the evaluation methodology, then analyze the evaluation results.

5.1 Evaluation Methodology

We chose some workloads from Phoenix [20] and PARSEC [3] benchmark suites, which can be built and run with Pthreads or PAthreads. Table 1 lists some detailed information about these workloads, where the first six workloads come from Phoenix, and the last four come from PARSEC. Columns 2–5 are profiling data collected by running each workload with 32 threads using the input dataset size given in Column 6, where *lock*, *wait*, *signal*, *barrier* in the title of Columns 2–4 represent numbers of `pthread_mutex_lock`, `pthread_cond_wait`, `pthread_cond_broadcast/signal`, and `pthread_barrier` function calls; *heap* refers to total bytes allocated in the heap segment. From the table, we find that only `pca`, `flui` and `dedup` take advantage of Pthreads synchronization function calls in application code. `dedup` is a typical VM-intensive application that consumes a lot of heap space and invokes many synchronization operations. PAthreads does not support Ad Hoc synchronizations [22] and stack write-sharing like Pthreads-compatible multithreading systems such as Dthreads [17], so it cannot support workloads such as `x264` from PARSEC.

Table 1. Benchmarks and their profiling data when running with 32 threads.

Benchmark	Lock	Wait/signal	Barrier	Heap (B)	Dataset
string_match (`sm`)	0	0	0	8243	500 MB
histogram (`hist`)	0	0	0	3014	1.4 GB
linear_regression (`lr`)	0	0	0	208	500 MB
matrix_multiply (`mm`)	0	0	0	4000256	2000
kmeans	0	0	0	2420832	100000
pca	2016	0	0	32040384	4000
swaptions (`swap`)	0	0	0	5431	Large
fluidanimate (`flui`)	22949710	1756/80	3411	36001	Large
blackscholes (`black`)	0	0	0	678	Large
dedup	258340	1044/69485	0	886135061	Large

Evaluation Methodology. We conducted all workloads on a 32-core Intel 4× Xeon E7-4820 system equipped with 128 GB of RAM. The OS is 64-bit executable Ubuntu 12.04 with kernel 3.2.0 and 4.10. All benchmarks and shared dynamic libraries are compiled by GCC v4.6.3 with optimization flag `-O3`. We logically disable CPU cores by Linux's CPU hotplug mechanism, which allows to disable or enable individual CPU cores by writing 0 or 1 to a special file (/sys/devices/system/cpu/cpuN/online), and the number of workers equals to the number of CPU cores enabled [23]. Each workload is executed 10 times. To reduce the effect of outlier, the lowest and the highest runtimes for each workload are discarded, thus each result is the average of the remaining 8 runs.

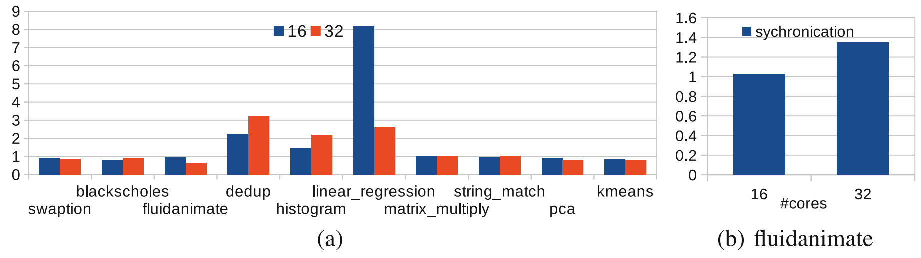


Fig. 6. (a) Runtime ratio: Pthreads relative to PATHREADS; (b) synchronization overhead ratio: PATHREADS relative to Pthreads for `flui`

5.2 Performance

We first compare the performance of PATHREADS with Pthreads. Figure 6(a) shows the runtime ratio of Pthreads relative to PATHREADS for each workload running with 16 and 32 cores, respectively. PATHREADS outperforms Pthreads in 3 workloads, including `hist`, `lr` and `dedup`. Two VM-intensive workloads built with PATHREADS run faster than with Pthreads, e.g., reaching up to 2.17× and 3.19× faster than Pthreads for `hist` and `dedup` on 32 cores, respectively. Although `lr` is not VM-intensive, PATHREADS greatly improves its performance, e.g., running 8.5× faster than Pthreads on 16 cores, due to eliminating false sharing. Reasons for this improvement are analyzed in Sect. 5.3.

Workloads `mm` and `sm` with PATHREADS show similar performance as those with Pthreads. Compared to Pthreads, however, PATHREADS has no advantage for the rest five workloads, where `flui` contains many Pthreads synchronization calls unlike the others, i.e., `swap`, `black`, `pca` and `kmeans`. For `flui`, PATHREADS cannot obtain the performance improvement on 32 cores, because the synchronization algorithm of mutex, barrier and so on, brings some performance overhead. As shown in Fig. 6(b), the synchronization overhead from PATHREADS is much higher than that from Pthreads. For the other four workloads, all of them have smaller calculation, so the overhead of thread creation has great impact on the program performance. Pthreads calls `clone()` to create child thread, while PATHREADS calls `fork()`. However, the overhead from `fork()` is higher than `clone()` in many different cores.

5.3 Performance Analysis in Detail

We further use Linux Perf to deeply analyze the reason why PATHREADS runs faster than Pthreads for `hist`, `dedup` and `lr`.

Locking Overhead in Linux Kernel. As analyzed in Sect. 2, hotspot kernel functions related to contention in read mode or write mode are different. The performance bottleneck of `hist` is contention on *mmap_sem* in read mode. When running `hist` with PATHREADS, overheads of two relative hotspot functions `down_read_trylock()` and `up_read()` are significantly reduced, both less than 0.5%

of the total execution time. But for `dedup`, the performance bottleneck is contention in write mode, and the relative hotspot function is `ticket_spin_lock()`. PATHREADS can also significantly reduce the overhead of spinlock for `dedup` to less than 2.37% on 32 cores. Significant reduction in kernel lock overhead indicates that per-thread isolated address space enabled by PATHREADS can effectively reduce contentions in Linux kernel.

More Linux Perf Data Analysis. Linux perf provides some underlying indicators, such as task-clock (TC), context-switch (CS), instructions per cycle (IPC) and stalled cycles per instruction (SCPI), to describe the internal execution of the application. TC represents the time spent on calculation. SCPI indicates the number of empty clock cycles, which can be caused by many reasons, such as cache, physical memory [1,8].

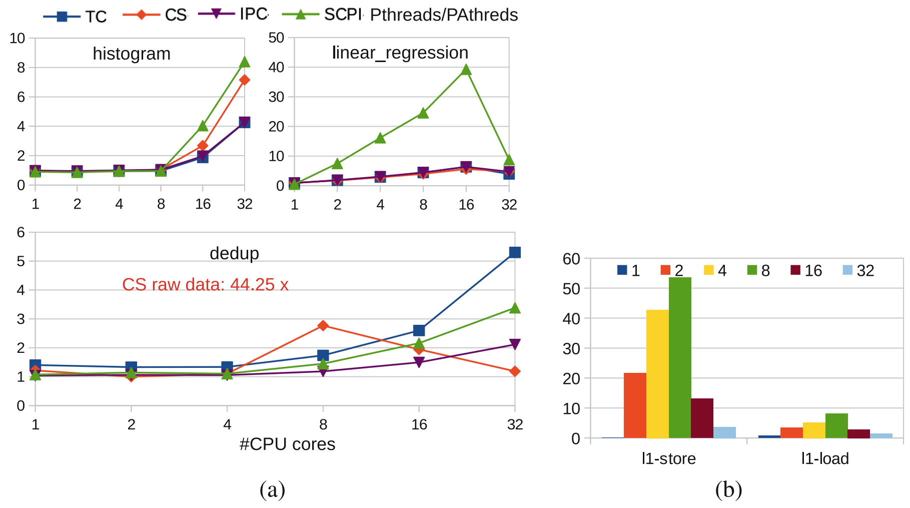


Fig. 7. (a) The ratio of L1 cache miss rate for `lr`: Pthreads/PATHREADS; (b) perf data about underlying execution: ratio - Pthreads/PATHREADS

By analyzing results collected by Perf, we find `hist` and `dedup` built with Pthreads generate large number of SCPI, due to the serious contentions on shared *mmap_sem*; while `lr` built with Pthreads has poor cache performance, due to false sharing.

We further compare the value ratio of Pthreads relative to PATHREADS. As shown in Fig. 7(a), for `hist`, the number of SCPI in Pthreads is 8× more than PATHREADS on 32 cores. `hist` with Pthreads suffers from serious lock contention, which generates many cycles stalled, leading to lower CPU utilization. The CPU utilization is greatly improved on PATHREADS since IPC on PATHREADS is 4× more than Pthreads.

For `dedup`, as core count increases, the ratios of IPC and SCPI (Pthreads/PATHREADS) also increase. Serious contentions on *mmap_sem* result in lower execution efficiency and higher memory latency for Pthreads. Especially, the ratio of context-switch is very high, where Pthreads is 132× higher than PATHREADS (CS ratios in Fig. 7(a) should be magnified by 44.25 times). Due to reducing contention in `dedup`, PATHREADS greatly enhances the efficiency of the task, where TC on Pthreads is 5× slower than PATHREADS.

For `lr`, the ratio of SCPI is extremely high, reaching 40× on 16 cores. Figure 7(b) further shows the ratio of cache miss rate (Pthreads/PATHREADS). We see that the load miss rate and store miss rate on Pthreads achieve 8× and 52× higher than PATHREADS, respectively. So the bad SCPI for `lr` is due to waiting for cache resources.

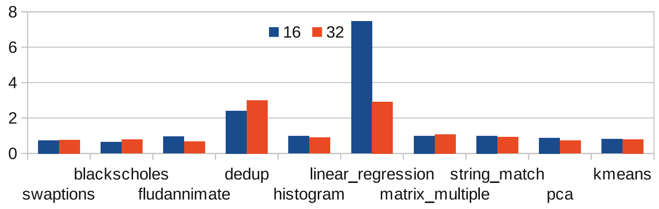


Fig. 8. Impact of Linux kernel 4.10: runtime ratio of Pthreads relative to PATHREADS

5.4 Impact of the Latest Linux Kernel

In this section, we examine the impact of Linux kernel optimization on the scalability and performance of Pthreads and PATHREADS by running them on Linux 4.10 kernel. Figure 8 shows the runtime ratio results that Pthreads compares to PATHREADS for each workload on different core counts. For two VM-intensive workloads, `dedup` still gets a bigger performance boost, reaching 2.99× faster than Pthreads on 32 cores, but `histogram` does not. By eliminating the false sharing, PATHREADS still greatly improve the performance of `lr`, reaching up to 7.45× faster than Pthreads on 16 cores.

To analyze the performance behavior in depth, we further use Linux Perf to collect information of hotspot functions for workloads. For `histogram` built with Pthreads, results show that `down_read_trylock` and `up_read` just occupy 0.17% and 0.13% of the total runtime on 32 cores. This indicates that the latest Linux kernel 4.10 has been able to effectively reduce the contentions in read mode, because since Linux kernel 3.15, address space competition has been reduced by optimizing VMA cache [11].

For `dedup`, the overhead of `__raw_spin_lock` in the 4.10 kernel is only 0.5% of the total time for Pthreads. This is because the 4.10 kernel uses a better queue spinlock [10] rather than the ticket spinlock. The hotspot functions for `dedup` with Pthreads on the 4.10 kernel are `flush_tlb_func()`, `__default_send_IPI_dest_field()`, occupying 3.46%, 2.26% of the total time, respectively. So frequent TLB refresh and communication between threads on *mmap_sem* still seriously affect the application performance on the latest Linux kernel.

6 Related Work

To avoid contentions caused by shared data structures, it is not uncommon for programmers to modify their applications, e.g., Psearchy from MOSBENCH [7] emulating threads by processes. However, it is difficult to modify the application from shared memory based to only private memory based. Furthermore, it cannot support legacy code directly.

Clements *et al.* [9] use super pages to improve scalability. On the X86-64, this can greatly reduce the number of page faults, dropping the contention on the read/write semaphore that manages address space. However, many multi-threaded applications often map small virtual memory regions, which cannot benefit from this solution. Furthermore, as memory consumes and CPU core count grows, using super pages would result in the same scalability problem. [7] analyzes seven system applications running on Linux on a many core computer, and points out that all applications trigger scalability bottlenecks inside Linux kernel. Boyd-Wickizer *et al.* [6] point out shared data structures used in kernel limit the application performance. And they argue that application should control sharing: the kernel should arrange each data structure so that only a single processor need update it. Wentzlaff *et al.* [21] find that page fault handling in Linux does not scale beyond 8 CPU cores.

The shared address space has a cost, for example, kernel VM operations such as handing soft page faults, allocating VM regions via `mmap/sbrk`, can degrade the performance and scalability of applications. Clements *et al.* [9] propose a new design to increase the concurrency of kernel operations on a shared addressed space by exploiting RCU [18], so that soft page faults can both run in parallel with operations that mutate the same address space and avoid contending with other page faults on shared cache lines. But this way still cause contentions on the coarse-grained lock.

7 Conclusions

This paper presents a Pthreads-compatible thread library PAthreads and a shared heap allocator *IAmalloc*. Experimental results show that PAthreads can improve the performance of VM-intensive application compared with the Pthreads. Strategies such as allocating in thread-local subheap and pad allocation employed in *IAmalloc* can improve the locality and eliminate false sharing. By our evaluation, we also give some analysis about underlying execution and access features. In the future work, we will try to enhance PAthreads by reducing thread creation and synchronization overhead among separate address spaces.

Acknowledgment. This work was partly supported by the grants of National Natural Science Foundation of China (No. 61772487) and Anhui Provincial Natural Science Foundation (No. 1808085MF198).

References

1. Anderson, J.M., Berc, L.M., Dean, J., et al.: Continuous profiling: where have all the cycles gone? ACM Trans. Comput. Syst. **15**(4), 357–390 (1997). https://doi.org/10.1145/265924.265925
2. Baldassin, A., Borin, E., Araujo, G.: Performance implications of dynamic memory allocators on transactional memory systems. In: 20th PPoPPACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 87–96. ACM (2015). https://doi.org/10.1145/2688500.2688504
3. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC benchmark suite: characterization and architectural implications. In: 17th PACT International Conference on Parallel Architectures and Compilation Techniques, pp. 72–81, October 2008
4. Birrell, A.: Implementing condition variables with semaphores. In: Herbert, A., Jones, K.S. (eds.) Computer Systems: Theory, Technology, and Applications. Monographs in Computer Science, pp. 29–37. Springer, New York (2004). https://doi.org/10.1007/0-387-21821-1_5
5. Bolosky, W.J., Scott, M.L.: False sharing and its effect on shared memory performance. In: USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems, vol. 4, p. 3. USENIX Association, Berkeley (1993)
6. Boyd-Wickizer, S., Chen, H., Chen, R., et al.: Corey: an operating system for many cores. In: 8th OSDIUSENIX Conference on Operating Systems Design and Implementation, pp. 43–57. USENIX Association, Berkeley (2008)
7. Boyd-Wickizer, S., Clements, A.T., Mao, Y., et al.: An analysis of Linux scalability to many cores. In: 9th OSDIUSENIX Conference on Operating Systems Design and Implementation, pp. 1–8. USENIX Association, Berkeley (2010)
8. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. Int. J. High Perform. Comput. Appl. **14**(3), 189–204 (2000)
9. Clements, A.T., Kaashoek, M.F., Zeldovich, N.: Scalable address spaces using RCU balanced trees. In: 17th ASPLOS International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 199–210, ACM, New York (2012). https://doi.org/10.1145/2150976.2150998
10. Corbet, J.: MCS locks and qspinlocks, March 2014. https://lwn.net/Articles/590243/
11. Corbet, J.: Optimizing VMA caching, March 2014. https://lwn.net/Articles/589475/
12. Drepper, U.: Futexes are tricky, November 2011. https://www.akkadia.org/drepper/futex.pdf
13. Evans, J.: A scalable concurrent malloc(3) implementation for FreeBSD. In: BSDCan Conference, Ottawa, Canada, May 2006
14. Gloger, W.: Dynamic memory allocator implementations in Linux system libraries, May 2006. http://www.malloc.de/en/index.html
15. Kleen, A.: Linux multi-core scalability. In: Linux Kongress, October 2009
16. Lea, D.: Dlmalloc: a memory allocator (2012). http://g.oswego.edu/dl/html/malloc.html. Accessed 24 Sept 2012

17. Liu, T., Curtsinger, C., Berger, E.: DTHREADS: efficient deterministic multithreading. In: 23rd SOSPACM Symposium on Operating Systems Principles, pp. 327–336, October 2011
18. McKenney, P.E.: Exploiting deferred destruction: an analysis of read-copy-update techniques in operating system kernels. Ph.D. thesis, Oregon Health & Science University (2004)
19. de Melo, A.C.: Performance counters on Linux. In: Linux Plumbers Conference, September 2009
20. Ranger, C., Raghuraman, R., Penmetsa, A., et al.: Evaluating MapReduce for multi-core and multiprocessor systems. In: 13th HPCA IEEE International Symposium on High Performance Computer Architecture, pp. 13–24. IEEE Computer Society, Washington, DC, February 2007. https://doi.org/10.1109/HPCA.2007.346181
21. Wentzlaff, D., Agarwal, A.: Factored operating systems (fos): the case for a scalable operating system for multicores. SIGOPS Oper. Syst. Rev. **43**(2), 76–85 (2009). https://doi.org/10.1145/1531793.1531805
22. Xiong, W., Park, S., Zhang, J., Zhou, Y., Ma, Z.: Ad Hoc synchronization considered harmful. In: 9th OSDIUSENIX Conference on Operating Systems Design and Implementation, pp. 1–8. USENIX Association, Berkeley (2010)
23. Zhang, Y., Cao, H.: DMR: A deterministic MapReduce for multicore systems. Int. J. Parallel Prog. 1–14 (2015). https://doi.org/10.1007/s10766-015-0390-5