Towards Exploiting CPU Elasticity via Efficient Thread Oversubscription

Hang Huang[1], Jia Rao[2], Song Wu[1], Hai Jin[1], Hong Jiang[2], Hao Che[2], and Xiaofeng Wu[2]
[1]National Engineering Research Center for Big Data Technology and System
Services Computing Technology and System Lab, Cluster and Grid Computing Lab
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China
[2]The University of Texas at Arlington, USA
Email: {huanghang, wusong, hjin}@hust.edu.cn, {jia.rao, hong.jiang, hche, xiaofeng.wu}@uta.edu

ABSTRACT

Elasticity is an essential feature of cloud computing, which allows users to dynamically add or remove resources in response to workload changes. However, building applications that truly exploit elasticity is non-trivial. Traditional applications need to be modified to efficiently utilize variable resources. This paper explores thread oversubscription, i.e., provisioning more threads than the available cores, to exploit CPU elasticity in the cloud. While maintaining sufficient concurrency allows applications to utilize additional CPUs when more are made available, it is widely believed that thread oversubscription introduces prohibitive overheads due to excessive context switches, loss of locality, and contention on shared resources.

In this paper, we conduct a comprehensive study of the overhead of thread oversubscription. We find that 1) the direct cost of context switching (i.e., 1-2 μs on modern processors) does not cause noticeable performance slow down to most applications; 2) oversubscription can be both constructive and destructive to the performance of CPU caches and TLB. We identify two previously under-studied issues that are responsible for drastic slowdowns in many applications under oversubscription. First, the existing thread sleep and wakeup process in the OS kernel is inefficient in handling oversubscribed threads. Second, pervasive busy-waiting operations in program code can waste CPU and starve critical threads. To this end, we devise two OS mechanisms, *virtual blocking* and *busy-waiting detection*, to enable efficient thread oversubscription without requiring program code changes. Experimental results show that our approaches can achieve an efficiency close to that in under-subscribed scenarios while preserving the capability to expand to many more CPUs. The performance gain is up to 77% for blocking- and 19x for busy-waiting-based applications compared to the vanilla Linux.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HPDC '21, June 21–25, 2021, Virtual Event, Sweden
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8217-5/21/06…$15.00
https://doi.org/10.1145/3431379.3460641

KEYWORDS

Elasticity; Over-threading; Container; Scheduling; Performance.

ACM Reference Format:
Hang Huang, Jia Rao, Song Wu, Hai Jin, Hong Jiang, Hao Che, and Xiaofeng Wu. 2021. Towards Exploiting CPU Elasticity via Efficient Thread Oversubscription. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '21), June 21–25, 2021, Virtual Event, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3431379.3460641

1 INTRODUCTION

Elasticity, commonly considered as one of the central attributes of cloud computing, is the ability of a system to adapt to workload changes by provisioning and deprovisioning resources in an automatic manner [18]. The goal of elastic resource allocation is to match system capacity with users' demand – resources should be timely scaled up to handle traffic spikes and accurately scaled down to allow users to pay only for what they need. Although existing virtualization techniques, such as *virtual machines* (VMs) and containers, support on-the-fly resource reconfiguration, truly exploiting resource elasticity remains a challenge. Traditional applications that assume constant resource availability may not benefit from resource elasticity, and can perform poorly under variable resource availability.

Among reconfigurable cloud resources, such as CPU, memory, and storage spaces, CPU is most suitable for elastic resource management. As CPU is mainly an execution vehicle with a small amount of data storage (i.e., registers and caches), it can be quickly provisioned or deprovisioned without loading or saving much of the application state (context). In contrast, memory or storage needs to be reclaimed before it can be re-allocated to other users. Therefore, popular virtualization platforms, such as VMware [30], Xen [2], KVM [24], and Docker [5], allow for realtime reconfiguration of the number of allocated CPUs without restarting a VM or container.

However, developing applications that can effectively utilize a varying number of CPU cores during its execution is non-trivial. On the one hand, an application needs enough concurrency (i.e., number of threads) to embrace increased hardware parallelism as the number of CPUs scales up. Intuitively, an application should have at least as many threads as the number of available CPUs to maintain high CPU utilization. On the other hand, when CPUs scale down, maintaining more threads than CPU cores, i.e., thread oversubscription, is believed to incur large overhead due to excessive context switches, loss of locality, and contentions for shared resources.

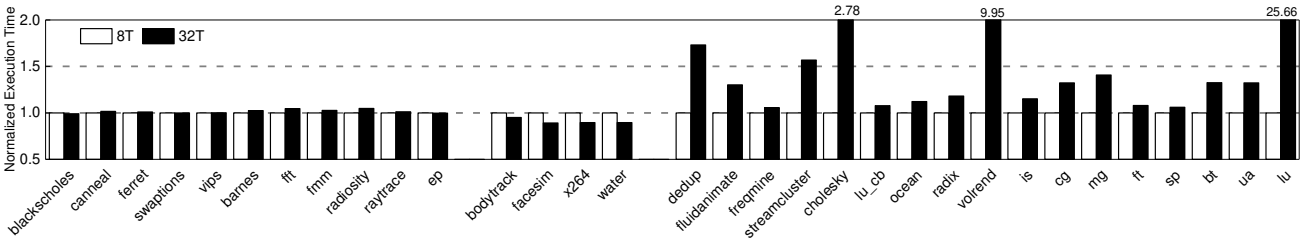


Figure 1: Performance of multithreaded programs with and without thread oversubscription

Existing efforts focused on dynamically adapting the number of threads in response to shared resource contentions and changes in system load. Thread adaptation has been implemented in user-level thread management [7, 34], runtime systems [8, 20], and compilers [25]. A common drawback of these approaches is that applications need to be instrumented to enable dynamic threading at runtime. Other approaches suspend and wake up threads from a thread pool to adjust thread count. For example, OpenMP [10] separately determines the number of threads for each parallel region; the HotSpot JVM [28] adjusts the number of active *garbage collection* (GC) threads before each GC starts. However, dynamic threading requires that workloads be dynamically distributed to threads since not all threads are active at a time. This imposes restrictions on how a program can be developed, e.g., using static or dynamic task-to-thread assignment, in order to utilize dynamic threading.

In this paper, we explore thread oversubscription as a general approach to exploiting CPU elasticity. Users always launch an optimal number of threads for an application regardless of the availability of CPUs in the cloud. The optimal degree of concurrency is determined offline as the maximum number of threads an application can use until it stops scaling. Provisioning sufficient concurrency allows the application to fully exploit hardware-level parallelism as CPUs scale. The **key challenge** is how to *efficiently execute oversubscribed threads when application concurrency does not match hardware parallelism*. Ideally, application performance with thread oversubscription should be close to that without oversubscription, incentivizing users to provision enough concurrency for elasticity. However, applications may suffer significant performance degradation when oversubscribing threads. Figure 1 shows how different multi-threaded programs perform with and without thread oversubscription. The benchmarks were from PARSEC [3], NPB [33], and SPLASH2 [36], and tested with two settings – 1) 8 threads on 8 cores, an exact one-to-one thread-to-core mapping; 2) 32 threads on 8 cores, an oversubscription ratio of 4.

As shown in the figure, the benchmarks can be classified into three groups. The first and second groups from the left include benchmarks that do not suffer or even benefit from thread oversubscription. In contrast, benchmarks from the third group suffer as much as 25x performance slowdown with oversubscription. The results suggest that thread oversubscription is already a viable approach to exploiting CPU elasticity for certain applications despite concerns about the potential overhead. However, thread oversubscription is still inefficient for a large number of applications. While these benchmarks are conventional parallel benchmarks, they involve various synchronization patterns and exhibit different levels of inter-thread coupling. Thus, the results shed light on how modern cloud workloads, which are commonly loosely-coupled, scale-out applications, such as key-value caches, databases, and machine learning analytics, would behave under thread oversubscription.

This paper presents a systematic study of the overhead and inefficiencies of thread oversubscription. The key findings are:

- The direct cost of oversubscription – thread *context switching* (CS) (mainly due to user-kernel mode transition) is inevitable and incurs a relatively constant overhead of 1.5 μs, regardless of the number of threads or inter-thread synchronization. Therefore, as long as applications do not perform context switching (either voluntarily due to synchronization or involuntarily due to CPU scheduling) too frequently, the overhead is negligible. Most applications meet this criteria and do not suffer noticeable slowdown from oversubscription (i.e., the first two groups in Figure 1).
- The indirect cost of oversubscription, including the loss of locality and contention on shared resources, does not necessarily cause slowdowns. Assume a strong scaling scenario wherein the problem size is fixed and users alter the thread count, oversubscription does not affect the working set size but changes the pattern of data access, which can be destructive or constructive. Although oversubscription affects the sequentiality of cache access and weakens the effectiveness of the hardware prefetcher, it helps improve the effectiveness of TLB. Our empirical study suggests that for most access patterns the constructive effect outweighs the destructive effect. Furthermore, oversubscription does not increase contention because the number of active threads participating in shared resource contention is determined by the number of cores rather than the total thread count.
- Two previously understudied issues are responsible for the large performance slowdowns in programs with tightly coupled threads. First, the existing mechanisms for thread sleep and wakeup in blocking synchronization are inefficient under oversubscription. Second, spinning or polling employed in busy-waiting synchronization can lead to cascading performance loss and a significant waste of CPU cycles under oversubscription.

Based on these findings, we believe that thread oversubscription can be made practical in exploiting CPU elasticity. To this end, we develop two OS mechanisms to support efficient thread oversubscription. *Virtual blocking* (Section 3.1) is a new mechanism for blocking synchronization that eliminates the high overhead of thread

sleep/wakeup. It relies on CPU runqueue operations to emulate the effect of blocking without actually putting threads to sleep. *Busy-waiting detection* (Section 3.2) is a software-based, general method for eliminating futile spinning. It is effective for any type of spin implementations, whether in the user space, the OS kernel, or in a VM, container, or in a native Linux environment. Experimental results on micro-benchmarks and realistic workloads show that our approaches can greatly improve the efficiency of thread oversubscription to a level similar to that of undersubscription.

2 ASSUMPTIONS, BACKGROUND, AND MOTIVATION

In this section, we briefly review the assumptions we make on thread oversubscription, describe how oversubscribed threads are scheduled on a single core, and study the overhead of oversubscription. Without loss of generality, we focus our discussions on a Linux environment and Intel platform.

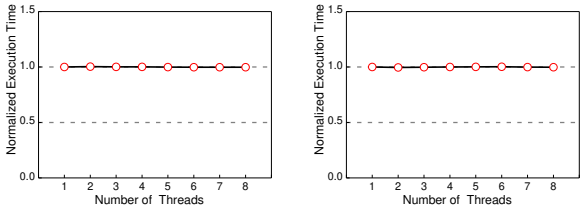
2.1 Assumptions

We assume that users are responsible for determining an appropriate number of threads for their applications, beyond which adding additional threads does not yield further speedup if more cores are provisioned. The users vary the number of threads for a fixed problem size (i.e., strong scaling) until the best performance is attained. Applications always run with the optimal number of threads, regardless of the availability of CPU cores. We consider threads are oversubscribed whenever the core count drops below the thread count due to dynamic resource management. To quantify the efficiency of thread oversubscription, we compare its performance with the case without oversubscription. For example, we measure the performance of an application running with 32 threads on 8 cores (oversubscription) and that of the same application running with 8 threads on 8 cores (baseline). If the two are close in performance, we consider oversubscription preferable as it allows for scaling as more cores become available. The applications considered in this work are shared-memory, multithreaded programs.

2.2 Scheduling Oversubscribed Threads

When threads are oversubscribed, it is certain that multiple threads from the same application reside on the same core. Therefore, it is important to understand how threads are scheduled in order to quantify the frequency of context switches. Modern OSes employ fair-sharing algorithms to schedule threads on the same core. Each thread is assigned a time slice during which a thread has dedicated access to the core. To improve responsiveness, modern OSes allow a thread to be preempted by a high-priority thread before finishing its time slice. An OS usually treats interactive threads, i.e., those frequently block and consume little CPU, as high priority threads. To prevent excessive context switching, OS imposes a minimum time slice before a thread can be preempted.

Context switches occur when threads are involuntarily or voluntarily descheduled. The completion of a time slice or the preemption by another thread forces the current running thread off the CPU. Therefore, involuntary descheduling happens on two occasions. For compute-bound workloads, threads are switched each time a regular time slice expires. For I/O-bound or blocking workloads, context switches occur no more often than the minimum time slice. In Linux's default *completely fair scheduler* (CFS), the regular time slice is 3 *ms*, and the minimum time slice is 750 μs. Voluntary switching happens when a thread yields CPU or blocks on an event (e.g., blocking on synchronization or waiting for I/O completion).



(a) Pure computation (b) Comp. with synchronization

Figure 2: Measuring the direct cost of context switching

2.3 The Overhead of Context Switching

The direct cost of context switching. While oversubscribing threads on a single core causes more context switches, its influence on application performance depends on the cost of each context switch and its frequency. We first measure the *direct cost* of a context switch, which includes the time spent on user-kernel mode switches (due to system calls or interrupts), CPU runqueue operations, and the load and save of thread contexts. We write a micro-benchmark that iterates for a fixed number of times. At each iteration, the benchmark performs some arbitrary computation. We started with one thread and increased the thread count across different runs. The iterations were evenly assigned to each thread emulating a strong scaling scenario in which the total problem size is fixed. We configure threads to yield CPU to trigger a context switch after they finish the minimum time slice (750 μs) in Linux. The benchmark is designed to have no data access.

Figure 2 (a) plots the execution times of this benchmark with different numbers of threads. Performance is normalized to that with a single thread. Since the benchmark does not have any memory access, the performance difference is due to the direct cost of context switches. We calculate the cost per context switch by dividing the difference of the overall execution time by the number of context switches. Results show that the per context switch cost is relatively stable at 1.5 μs on our Intel-based platform (see Section 4 for details on the hardware configuration). Since for compute-bound workloads, each thread is guaranteed a minimum time slice (750 μs), and there is one context switch per 750 μs, the overall cost to the benchmark execution time is only 0.2%, and it does not increase with the number of threads. Note that our benchmark is different from the one used in [26], which measured the cost of process context switching by having two processes communicate via two pipes. First, process context switches are much more expensive due to address space switching. Second, pipe communication triggered context switches require threads to sleep when switched out. As will be discussed later, the thread sleep/wakeup process is a major source of inefficiency in thread oversubscription.

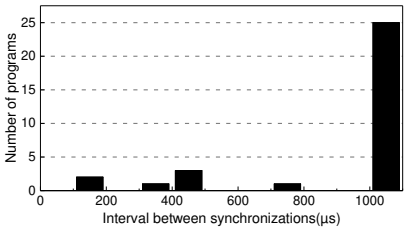


Figure 3: The interval between synchronizations in the PARSEC, SPLASH2, and NPB benchmarks

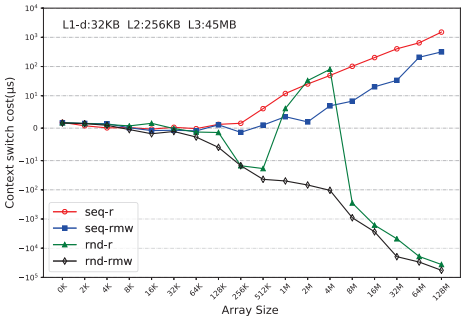


Figure 4: The indirect cost of context switches

For workloads that block (voluntary switch), e.g., threads blocked waiting for a lock, the frequency of context switching depends on the interval between synchronization, which is not controlled by the CPU scheduler. The frequency of synchronization increases with the number of threads because the work that should be done before each synchronization is less with more threads. Figure 3 shows the interval between two synchronizations in the PARSEC, SPLASH2, and NPB benchmarks. All benchmarks are configured with their respective optimal number of threads (i.e., 16 or 32 threads) on our platform. Since not all (only failed) synchronizations lead to context switches, the frequency of context switching is no higher than synchronization frequency. Figure 3 suggests that most applications perform context switches no more often than every 1000 μs, resulting in a CS overhead of 0.15% compared to the corresponding sequential program. The most frequent context switch could occur in *facesim* which has a synchronization interval of 160 μs. Nevertheless, the CS overhead is still less than 1%.

The indirect cost of context switching. Besides the time to perform a switch (direct cost), there are other performance penalties (*indirect cost*) associated with context switching. Compared with solving a problem sequentially (one thread), dividing the problem among multiple threads (concurrency) with no real parallelism (oversubscribed threads time-share the same core) has many implications on cache performance and shared resource contention. One concern about over-threading on multicore processors is that a large number of threads would lead to high contention on locks or shared cachelines. We show that this is not an issue for thread oversubscription, in which the number of actively running threads is at most the number of cores. We modify the micro-benchmark to update a variable shared by all threads at each iteration by atomic instruction `__sync_fetch_and_add`. This change would incur heavy cache coherence traffic on multiple cores. However, as shown in Figure 2 (b), oversubscription does not add additional overhead or indirect cost to the benchmark because of synchronization.

The effect of context switching on cache performance is more intricate. We modified the micro-benchmark so that each thread traverses a sub-array between context switches. The total size of all sub-arrays is fixed (strong scaling). By varying the total array size, we alter the working set size of the program. Each element in the array is a double number (8 byte), and thus 8 elements take one cache line. Figure 4 shows the indirect cost of context switches for two threads with four access patterns. The indirect cost is calculated as $\frac{t_{over}-t_{serial}}{\#\ of\ CS}$, where t_{over} and t_{serial} are the execution times with two threads and one thread, respectively. All threads were pinned to the same core. A negative cost indicates that thread oversubscription helps improve performance. The four access patterns are: 1) sequential read (*seq-r*), 2) sequential read-modify-write (*seq-rmw*), 3) random read (*rnd-r*), and 4) random read-modify-write (*rnd-rmw*).

As shown in Figure 4, running two threads, each sequentially accessing the arrays, incurred increasing indirect cost of context switching as the working set size increased. The performance penalty started to climb at a total array size of 512KB, at which each thread's working set size (256KB) can barely fit in the L2 cache. The main reason for the penalty is the loss of sequentiality when accessing array elements from two threads. The hardware prefetcher is more effective for a single thread as its access pattern is more predictable. The trend of *seq-rmw* is similar to that of *seq-r*. With an array of 128MB, the indirect cost of context switch is around 1 *ms*, more than 600x of the direct cost. At the size of 128MB, without any computation on the data, each thread needs 17.5 *ms* to access their sub-arrays before context switching. Therefore, the overhead due to the indirect cost is less than 6% ($=\frac{1ms}{17.5ms}$).

In contrast, thread oversubscription could improve the effectiveness of TLB for random access. Figure 4 shows that randomly reading the array from two threads led to a clearly negative cost starting at 256KB. Between size 1MB and 4MB, the cost climbed to positive. After that running with two threads consistently outperformed that with one thread. It is important to review the array's memory allocation to understand the performance trend. The array was allocated by the parent thread and partitioned into two sub-arrays for the two threads. This is a common way for memory allocation in multi-threaded programs. The single-threaded program randomly accesses the entire array while each thread in the two-thread program takes turns to randomly access half of the array. The number of address translations needed to access the array is $\frac{array\_size}{page\_size}$. The default page size is 4KB. When the number of address translations in the array is larger than the number of entries in TLB, each random access to the entire array would almost certainly need a new page address, which causes a high TLB miss rate. By dividing the array into two, address translations for each sub-array may fit in the TLB. While randomly traversing a sub-array will initially cause TLB misses, all subsequent accesses would be TLB hits since the TLB can hold all page addresses the sub-array needs. On the other hand, when the per-thread sub-array size is larger than the size of the L2 data cache but smaller than the L3 size, i.e., total array size between

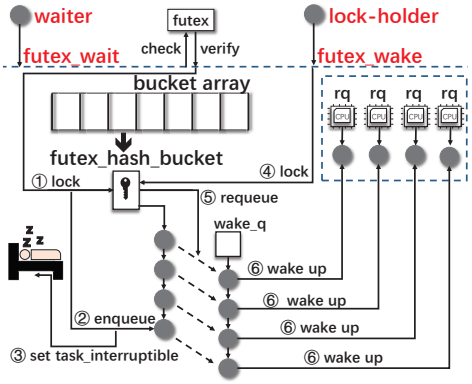


Figure 5: The process of thread sleep/wakeup with *futex*

512KB and 64MB, random access by a single thread leads to higher L2 performance. Since the sub-array (half of the total array) cannot fit in the L2 cache, switching between threads certainly results in all L2 misses and a flush of the L2 cache.

Our testbed has two Intel processors (Xeon E5-2695) equipped with a two-level TLB. The first level data TLB has 64 entries and the second level has 1536 entries. Therefore, the two-level TLBs can address 256KB (4KB × 64) and 6MB (4KB × 1536) of data, respectively. As shown in Figure 4, at array size 256KB and 512KB, the benefit of fitting sub-array addresses in the first level TLB outweighed the effect of the L2 data cache. Between array size 1MB and 4MB, neither the sub-array nor the total array can fit in the first-level TLB, but both can fit in the second-level TLB. Thus, oversubsciption does not help TLB performance but suffers more L2 misses. In this region, running with fewer threads would be more favorable. Beyond 4MB, only the sub-arrays can fit in the second-level TLB, running with more threads becomes more favorable. For read-modify-write, the L2 cache is not an important factor as dirty cache lines need to be written back to the L3 cache or memory. Note that the benefit of TLB performance gain is an order of magnitude higher than that of the L2 cache. Therefore, it is always more favorable to oversubscribe threads for RMW workloads with random access.

Realistic workloads are a combination of sequential and random accesses, and contain a mix of reads and writes. *Contrary to what was believed previously, the results suggest that thread oversubscription should benefit caching performance in most cases with a worst-case performance penalty of 6%.*

2.4 Inefficiencies in Managing Oversubscribed Threads

In this section, we show that mechanisms designed for managing threads on multiple cores are inefficient for managing oversubscribed threads on a single core. The inefficiencies are unique to parallel programs with inter-thread synchronization. In what follows, we examine the mechanisms for blocking and busy-waiting synchronization, and identify the root causes that are responsible for the large performance slowdowns.

Blocking synchronization, such as *mutex*, *semaphore*, and *condition variable*, puts a caller thread into sleep if it fails to acquire a lock or does not meet a certain condition. The thread is later woken up when the lock becomes available or the condition is met. Another mechanism for blocking synchronization is event-based asynchronous I/O, such as Linux *epoll*. Threads sleep on a list of file descriptors and wake up when events are posted on the files. Compared with busy-waiting synchronization, which continuously tests a lock or a condition, blocking does not waste CPU cycles.

As thread sleep/wakeup needs to be handled by the OS kernel, blocking synchronization is more expensive, requiring trapping into the kernel and thread state transitions. Without loss of generality, we use the design of *futex* to illustrate how Linux handles thread sleep and wakeup (as shown in Figure 5). *Fast userspace mutex* (futex) is a low-level interface in the OS kernel for implementing blocking synchronization. The futex itself is a variable at the user level. Successful synchronization, e.g., lock acquisition, returns directly from the user space. Unsuccessful synchronization traps into the kernel and goes through the steps depicted in Figure 5.

As shown in Figure 5, upon failing to acquire the lock, the waiter thread invokes `futex_wait` and starts the sleep process. It first acquires the lock that protects the `futex_hash_bucket` queue, where it will be sleeping. The waiter is then removed from the CPU runqueue, enqueued on the sleep queue, and its runtime state is changed from "runnable" to "sleep" (`TASK_INTERRUPTIBLE`). When the user-level lock is released, the lock holder acquires the lock on the hash bucket and moves one or more waiters from the sleep queue to a temporary wakeup queue `wake_q`, from where the waiter(s) awaken. This design is to prevent holding the bucket lock for too long as thread wakeup can take long. The lock holder is responsible for awakening the waiters one at a time, which includes selecting a core for this waiter and enqueuing it to the new runqueue.

The sleep and wakeup process can be quite expensive when threads are oversubscribed for the following reasons:

Complex wakeup process. While thread sleep is not on the critical path, the wake up process can significantly delay program execution. The wake up operation (step 6 in Figure 5) first selects the most idle core and inserts the awakening thread to its runqueue. This operation requires locking the runqueue of the new core. After that, the kernel checks if the waking thread should preempt the current running thread on the newly selected CPU. When there are many more threads than cores and a large number of threads are awakening, runqueue locking and the possible preemption of a recently awakened thread not only cause serialization but also incur cascading performance degradation. Moving waiters from the bucket queue to the temporary `wake_q` queue adds additional serialization.

Fluctuating load and unnecessary migrations. Thread sleep and wakeup involve state transitions between "runnable" and "sleep". Runnable threads are calculated as active load on a CPU. Linux performs thread migration to balance load across cores if significant imbalance is detected. When a large number of threads do blocking synchronization, frequently switching between "runnable" and "sleep", the load on each core can fluctuate wildly, triggering excessive, unnecessary migrations.

Busy-waiting synchronization. Compared with blocking synchronization, busy-waiting synchronization (spinning) provides fast lock acquisition at the cost of wasting CPU cycles on spinning. The wasted CPU time is not crucial to program performance if each thread runs on a dedicated core because the core would otherwise be

```
Pthread_spin_lock--glibc
/*** Program Code ***/
do{
    atomic_spin_nop ();
    val = atomic_load_relaxed (lock);
}while (val != 0);
/*** Assembly code ***/
.L2:
    movl  $0, %eax
    call   atomic_spin_nop@PLT
    call   atomic_load_relaxed@PLT
    cmpl  $0, -20(%rbp)
    jne    .L2
```

```
Sync_right--lu
/*** Program Code ***/
do while (isync(iam) .eq. 1)
   !$omp flush(isync)
end do

/*** Assembly code ***/
.L10:
    cmpl   $1, %eax
    jne     .L9
    jmp    .L10
```

Figure 6: Various spin implementations

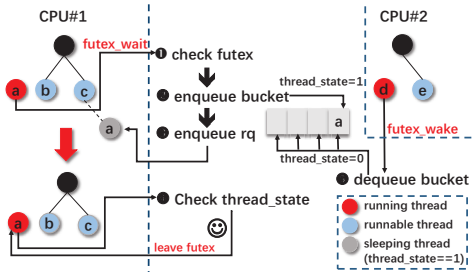


Figure 7: Virtual blocking in *futex*

idle. However, when threads are oversubscribed and spinning threads are placed with other threads doing useful work, spinning can burn the CPU time that can be used for useful work. Oversubscription also exacerbates the *lock-holder preemption* (LHP) problem. A waiter thread may exhaust its time slice doing spinning before becoming the lock holder who will soon be preempted.

An intuitive solution is to stop spinning threads whenever excessive spinning is detected. Towards this goal, software and hardware-based approaches have been developed. Software approaches [12, 21, 31] employ the spin-then-block strategy that stops spinning if a predefined threshold on spin time is reached. However, these approaches require application source code change to enable the hybrid waiting policy. Hardware approaches, such as Intel *pause-loop-exiting* (PLE) and AMD *pause filter* (PF), detect common patterns in spin loops from hardware events and stop threads which are identified as spinning. Specifically, PLE and PF detect the execution of the `PAUSE` or `NOP` instructions, a building block of many spin implementations. For example, in Figure 6, the code on the left (*pthreads* spin-lock) includes the `NOP` instruction in the spin loop. Unfortunately, both PLE and PF are designed for virtualized environments and can only detect spinning in *virtual CPUs* (vCPUs). Furthermore, there are a variety of spin implementations that do not include special hardware instructions. As shown in Figure 6, the spin loop in the *lu* benchmark (from the NPB benchmark suite) is simply a busy loop continuously testing a variable. For this spin implementation, neither PLE nor PF is effective.

Summary Our analysis has shown that thread oversubscription should introduce no noticeable performance slowdown to realistic workloads. The direct and indirect costs of context switching are negligible or even beneficial to most programs without inter-thread synchronization. The large slowdowns we observe are due to the inefficiencies that arise in the OS kernel when managing a large number of threads on a single core.

3 DESIGN AND IMPLEMENTATION

In this section, we present two mechanisms to address the inefficiencies of thread oversubscription. *Virtual blocking* (VB) is a new method for implementing blocking synchronization in the OS kernel. It manipulates the scheduling of threads to emulate the effect of sleep and wakeup, while preserving the properties of the original blocking synchronization. VB is transparent to user-level locking and requires no changes to users' code. *Busy-waiting detection* (BWD) is a software-based approach for detecting spin loops. It periodically examines the *last branch records* (LBRs) and looks for patterns that are common to various spin implementations. If spinning is detected, BWD instructs the CPU scheduler to immediately deschedule the spinning thread to prevent wasting CPU cycles.

3.1 Virtual Blocking

The sleep and wakeup process in Linux requires threads to be moved between sleep queues and the CPU run queue. Multiple queue locking and thread state transitions are needed during the process. In an oversubscribed scenario, these operations become especially expensive. When multiple threads wake up from the same core, not only is lock contention more intense, but waking threads are also more likely to be moved away from the core they ran before sleeping. The essence of blocking is to exclude sleeping threads from running on the CPU. Virtual blocking emulates the effect of sleeping by skipping blocked threads in scheduling. As such, the sleep queues are entirely removed, thereby eliminating much of the overhead during thread wakeup. While VB is a general approach applicable to any thread blocking mechanism, we present its design in the context of `futex` and Linux CFS scheduler. We discuss how to implement VB in event-based blocking mechanism *epoll* in Section 4.2.

VB adds a flag `thread_state` to each thread to indicate whether the thread is blocked (1) or not (0). Blocked threads are skipped during CPU scheduling until `thread_state` is cleared. Figure 7 shows how VB is integrated with futex. VB still preserves the `futex_hash_bucket` queue in futex in order to preserve the order threads is put to sleep as well as their wakeup order. However, threads are never moved from `futex_hash_bucket` to the CPU run queue. When a thread is awakened by `futex_wake`, its `thread_state` is cleared and it is removed from the `futex_hash_bucket` queue. The blocked thread is then moved to the end of the CPU run queue. This design ensures that the blocked threads never get a chance to run as long as there is at least one runnable thread whose `thread_state` is 0, on the same CPU run queue. If all threads on a core are blocked, which is not uncommon under thread oversubscription, each thread takes turns to briefly run on CPU to check if its flag has been cleared. After `thread_state` turns to 0, a thread resumes normal scheduling and is "awakened" from virtual blocking.

Since threads are not put into real sleep, an important change in futex is needed. Instead of yielding CPU after a thread is enqueued to the bucket queue, the thread is kept active and continuously checks the value of `thread_state`. This is equivalent to spinning on the flag and handing over to the CPU scheduler for managing sleeping. Note that the change does not affect the semantics or the interface of futex to userspace applications or libraries. For example, no changes

in *pthreads* are needed to use the new futex. Additionally, the spinning on the flag in futex does not cause a waste of CPU cycles. Recall that threads in virtual blocking are not scheduled if there are other runnable (non-blocked) threads on a CPU, they are unable to spend CPU time on spinning. In the case that all threads are blocked, the spinning are not wasteful since no other useful work or runnable threads can be scheduled.

While the bucket queue, from where a locking algorithm decides to which thread the lock should be granted, preserves the original order of sleep and wakeup, we further modify the CPU scheduler to immediately schedule threads that are waking from virtual blocking, in a similar way the traditional wake up process prioritizes those waking from real sleep. We also devise a mechanism to disable VB if threads are not oversubscribed. If the number of threads waiting on the bucket queue is smaller than the number of cores, i.e., all waiting threads are able to obtain a dedicated core when simultaneously waking up, VB is turned off.

Implementation We implemented VB in the default Linux scheduler CFS. To ensure that blocked threads are skipped in scheduling, they are inserted to the tail of the *red-black* (RB) tree-based CFS run queue. As the RB tree is sorted by threads' virtual runtimes, blocked threads are assigned an arbitrarily large virtual runtime. Threads' true virtual runtimes are restored when they wake up from VB. `Thread_state` is an atomic variable in the `task_struct` structure to avoid introducing additional locking in futex.

3.2 Busy-waiting Detection

As previously discussed, spinning synchronization can cause cascading performance collapse among oversubscribed threads. BWD seeks to detect futile spinning in an application and deschedule the spinning threads so that CPU cycles can be spent on critical threads. Similar to other existing spinning detection mechanisms, BWD infers spinning threads from the host OS or hypervisor without requiring to instrument application source code. Although there are various kinds of busy-waiting implementations, they share a common feature – a spin loop is a small code segment that is repeated for many times. Specifically, 1) spin loops are typically backward conditional branches; 2) each iteration is quite short, taking only a few cycles to execute; 3) during spinning, branching is predictable with hundreds of thousands backward branching until moving onto the next code segment.

While non-spinning code segments may behave similarly to a spinning loop in some aspects, the combination of the three heuristics can reliably identify busy-waiting. BWD employs the the *last branch records* (LBRs) and hardware *performance counters* (PMCs) as well as a high resolution timer to detect spinning. Figure 8 illustrates the architecture of BWD. We configure a *high resolution timer* (`hrtimer`) [15] on each core to periodically check the LBRs and PMCs. If the readings of LBRs and PMCs indicate spinning, the interrupt handler of the timer forces the thread or vCPU currently running on the core to be descheduled. The LBRs record the `from` and `to` virtual addresses of the recently completed branches. Branches that are caused by function calls and returns are excluded from LBRs. This is to capture spin implementations that involve nested function calls (as shown in Figure 6). We further configure PMCs to record

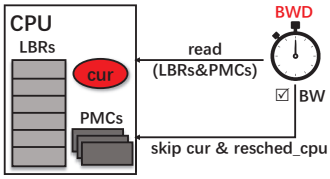


Figure 8: Busy-waiting detection based on LBR

the number of TLB misses and L1 data cache misses. All the LBR and PMC records are cleared for each monitoring period.

BWD identifies spinning if during the last timer interval 1) all branches recorded in LBRs (16 entries on our platform) were identical, backward branches, and 2) there were no TLB misses or L1 data cache misses. The timer interval is carefully set to 100 μs, the minimum interval that does not impose noticeable overhead. BWD requires that all the 16 entries be filled during an interval in order to identify a code segment as a spin loop. Recall that each iteration of a spin loop, which triggers a conditional branching, only takes a few cycles. It is almost certain that spin loops can fill all the 16 entries during the 100 μs interval. In comparison, non-spinning loops can be, on average, at most $\frac{100\mu s}{16} = 6.25\mu s$ long to fill all the entries. BWD further requires that there be no TLB misses or L1 data cache misses since the `from` and `to` addresses in spin loops are always cached in TLB and the data accesses in a tight spin loop should not miss any data caches.

On our Intel platform with the broadwell architecture, TLB has only 1600 entries, and L1 data cache is 32KB. We profiled all 32 benchmarks in PARSEC, NPB, and SPLASH-2, and found that on average 1) these programs retire 3000 instructions per microsecond; 2) every 45 instructions cause 1 L1 miss; 3) every 890 instructions cause 1 TLB miss. Therefore, they on average cause 6667 L1 misses and 337 TLB misses per BWD accounting period (i.e., 100μs). Based on the profiling, we believe that the combination of the three heuristics is a reliable metric for spin detection. Once busy-waiting is detected, BWD deschedules the spinning thread and sets a `skip` flag on the thread. This ensures that the spinning thread will not be scheduled until other threads on the same core are scheduled at least once, which helps schedule critical threads sooner to avoid future spinning.

We **implemented** VB and BWD in Linux kernel version 5.1.12. VB and BWD added 217 and 104 lines of code to the OS kernel respectively, and required no changes in user-space libraries or applications.

4 EVALUATION

4.1 Experimental Settings

Hardware. Our experiments were performed on a Dell T630 server, which was equipped with dual 18-core Intel Xeon 2.10 GHz processors with hyper-threading enabled, 128GB memory, and a 1TB SATA hard drive.

Software. We used Ubuntu 16.04 64bit and Linux kernel version 5.1.12 as the host OS. Docker 18.06.1 was used as the containers

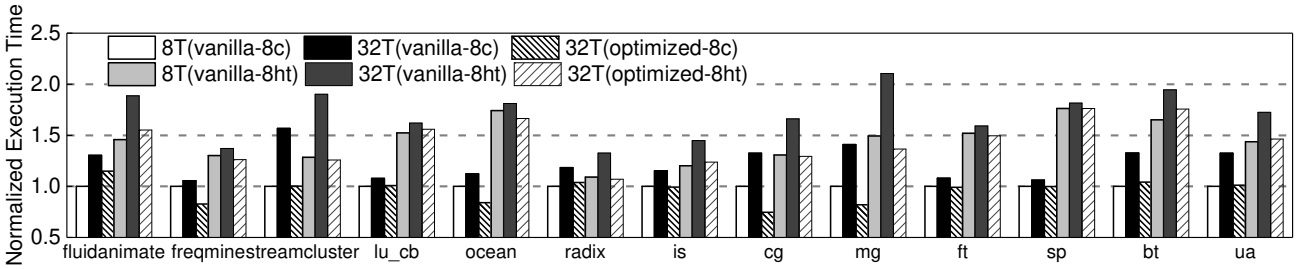
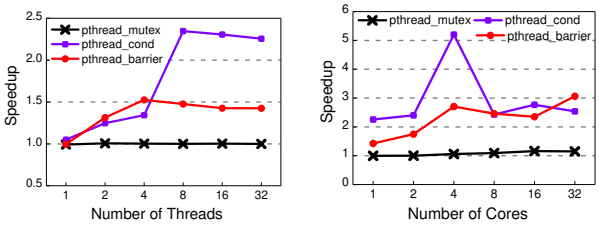


Figure 9: The performance improvement due to virtual blocking in parallel applications with blocking synchronization in 8 cores or 8 hyper-threads of 4 cores



(a) Varying threads on a single core (b) 32 threads on varying cores

Figure 10: The effect of virtual blocking on pthreads primitives

Table 1: The runtime statistics under thread oversubsciption

App	CPU utilization(%)			#In-node Migr			#Cross-nodes Migr		
	8T	32T	Opt	8T	32T	Opt	8T	32T	Opt
flu	734	701	782	45	98384	58	8	48835	26
freq	759	741	780	76	509	312	4	192	142
str	725	542	775	20183	672379	197	122	63250	211
lu_cb	567	552	581	52	7672	93	4	3203	18
ocean	677	664	763	127	107913	90	24	31809	44
radix	694	724	725	37	369	159	4	252	16
is	757	764	764	2	537	29	4	238	19
cg	667	674	797	444	55580	21	116	20582	24
mg	677	682	787	9	25256	16	4	3311	17
ft	754	758	793	3	2702	35	8	1534	32
sp	796	728	799	18	55844	48	4	24582	50
bt	786	771	799	24	22225	38	12	11123	50
ua	776	622	799	83	516791	60	24	78537	84

technology and KVM was the virtual machine technology. Experiments were conducted on the *POSIX Threads* (pthreads) with the *GNU C Library* (glibc) 2.27 and OpenMP with GCC 7.4.0.

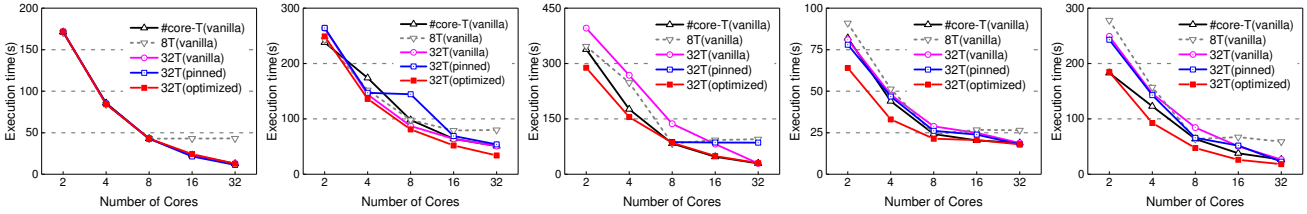
Benchmarks and methodology. We first created two micro-bench marks to evaluate the effectiveness of virtual blocking and busy-waiting detection. Then, we showed the performance of VB and BWD with the PARSEC 3.0 [3], SPLASH-2 [36], NAS parallel benchmarks [33], and Memcached under thread oversubscription. In all tests, the baseline performance was obtained in vanilla Linux with a one-to-one thread-to-core mapping. For example, 8T (vanilla) indicates 8 threads running on 8 cores. To measure the efficiency of oversubscription, we increased the thread count without adding more cores. On our platform, most applications can scale to 32 cores. If not otherwise stated, the maximum number of cores was 8 and the maximum number of thread was 32. Therefore, 32T (vanilla) indicates an oversubscription ratio of 4. The results with our proposed optimizations are labeled as 32T (optimized). Each result was the average of 10 benchmark runs.

4.2 Blocking Synchronization

Micro-benchmarks. We first evaluated the effectiveness of VB with a micro-benchmark, in which multiple threads repeatedly call pthreads blocking synchronization primitives for ten thousand times. Threads are synchronized with each other with mutex, barrier, and condition variable. Figure 10 shows how VB can improve the performance of blocking synchronization. Results are normalized to the performance of the vanilla Linux. Figure 10 (a) shows the results on a single core, in which the inefficiency of oversubscription mainly comes from the locking on the sleep queue and CPU run queue. It suggests that VB is most effective for group synchronizations, i.e., barrier and condition variable, in which multiple threads may simultaneously awake. The elimination of the wake up overhead in VB led to 1.52x and 2.34x speedup over Linux for barrier and condition variable, respectively. As shown in Figure 10 (b), the benefits of VB for group synchronization increased when 32 threads were running on multiple cores. The speedups rised up to 3x and 5x. In contrast, one-to-one synchronization, e.g., mutex, does not benefit much from VB. Since only one waiter thread is woken up when mutex is released, the original wake up process in Linux does not cause much inefficiency.

Conventional parallel applications using Pthreads. Next, we show how much benefit VB can bring to realistic benchmarks in oversubscribed scenarios. Compared to the micro-benchmark, whose execution time is dominated by synchronization, blocking synchronization only accounts for a small portion of runtime in real-world applications. We tested 27 benchmarks from PARSEC, SPLASH-2, and NPB. The selected benchmarks covered the three groups shown in Figure 1 that are not affected by, benefited from, or suffered from oversubscription. The unselected benchmarks (i.e., *dedup*, *cholesky*, *radiosity*) either cannot scale up to more than 8 threads or have a short and unstable execution time.

The benchmarks were run in containers configured with two settings: 8 cores or 8 hyperthreads on 4 cores. Due to space limit, Figure 9 only shows the results of application that are suffered from oversubcription. For applications that are not affected by or benefited from oversubscription, VB performed similarly to the vanilla Linux and introduced no more than 0.5% overhead. While experiments with and without hyperthreading achieved different performance, the trend was similar across all benchmarks. Thus, we focus our

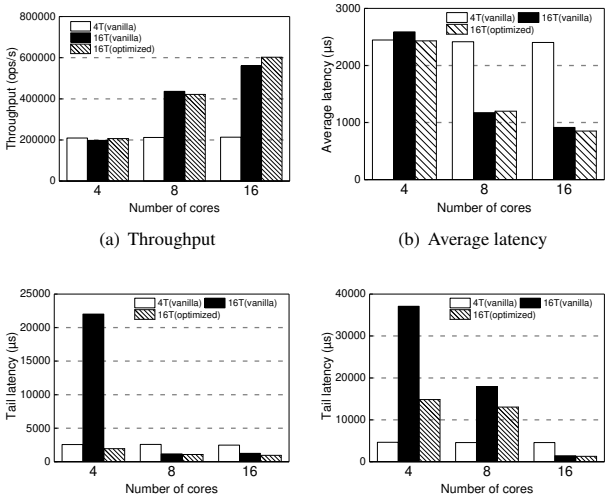


(a) ep (b) facesim (c) streamcluster (d) ocean (e) cg

Figure 11: The benefit of VB and oversubscription in five different kinds of applications

discussions on results with hyperthreading disabled. As shown in Figure 9, thread oversubscription introduced 5.5% to 56.7% performance slowdown under vanilla Linux. Table 1 shows that the culprits were the loss of CPU utilization and excessive thread migrations across cores. The low CPU utilization in vanilla Linux was due to the expensive wake up process, during which programs make no progress. In comparison, VB helped attain performance close to the baseline without oversubscription (i.e., 8 threads on 8 cores), except for *fluidanimate*. While *fluidanimate* under VB still outperformed that with the vanilla Linux, it suffered 17% degradation compared to the baseline. The reason is that the number of locks in *fluidanimate* scales with the thread count, thereby inevitably inflicting higher overhead with more threads. Notably, VB outperformed the baseline in *freqmine*, *ocean*, *cg*, and *mg* with an oversubscription ratio of 4, suggesting that VB could be an effective approach for improving the efficiency of blocking synchronization in non-oversubscribed scenarios. Table 1 confirms that VB greatly improved CPU utilization and reduced the number of thread migrations. It is worth noting that VB even helped reduce the number of migrations compared to 1-1 thread to core mapping case, e.g., in *cg* and *streamcluster*. In VB, "blocked" threads are skipped in either scheduling or migration while threads can still be migrated due to transient load imbalance in under-subscribed systems.

Runtime adaptation We then dynamically varied the number of available cores and evaluated how oversubscribed threads exploit CPU elasticity. We chose five benchmarks with distinct characteristics and allocated 8 cores at startup, while varying the number of cores from 2 to 32 at runtime. Since Table 1 suggests one of VB's benefits is to throttle thread migrations, we also evaluated how VB is compared to CPU pinning. The baseline was 8 threads. *Ep* did not suffer much from oversubscription and benefited greatly from it. Compared to provisioning 8 threads, *ep* with 32 threads can better utilize 32 cores, achieving a 51% performance gain. *Streamcluster*, *ocean*, and *cg* always suffered from oversubscription in vanilla Linux. With VB, running 32 threads was never worse than running 8 threads, and always better than with pinning method. *facesim* is a bit different. In some cases, it benefited from oversubscription even in vanilla Linux. VB was able to further improve its performance by as much as 34%, while pinning threads to cores can hurt performance as tasks are not evenly distributed among threads. There are two fundamental limitations of pinning: 1) it is unable to deal with varying CPU count and needs to re-pin threads when CPU count changes. In most applications we tested, programs crashed when CPU count decreased. 2) pinning turns off all migrations even real



(a) Throughput (b) Average latency

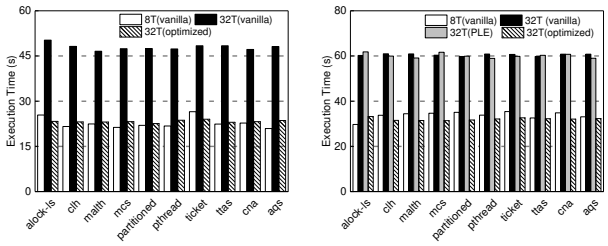
(c) 95^{th} Percentile Tail latency (d) 99^{th} Percentile Tail latency

Figure 12: The benefit of VB and oversubscription in a memcached server

load imbalance occurs while VB only prevents migration due to frequent sleep and wakeups. These results suggest that VB eliminates the inefficiencies of blocking synchronization and users are always encouraged to over-provision threads to exploit CPU elasticity.

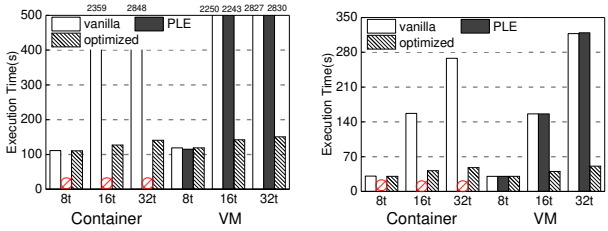
Cloud workloads using event-based asynchronous I/O. Compared to conventional parallel programs, cloud workloads usually employ loosely-coupled threads and event-based notification mechanisms for thread coordination. *Memcached* is a widely used high-performance, distributed memory caching system. It employs the *libevent* library, which is based on *epoll*, to synchronize worker threads. Initially, *memcached* worker threads call `epoll_wait` and block to wait for incoming requests. Workers are awakened upon the arrival of client requests. Besides epoll, memcached also relies on *pthreads* mutex to protect the hash table it uses for key-value pairs lookup from concurrent updates. Similar to the changes made in futex, we implemented VB in epoll by removing the sleep queue and emulating sleeping via schedule skipping. Since memcached uses both epoll and futex, VB was enabled for both blocking mechanisms.

We used *mutilate* to stress test the performance of the memcached server. The baseline had 4 worker threads while the oversubscribed



(a) Container (b) KVM

Figure 13: The applicability of BWD to various spinlocks



(a) lu (b) volrend

Figure 14: The effectiveness of BWD in user-customized spinning

scenarios had 16 threads. We evaluated three core settings: 4, 8, and 16 cores and with 16 workers the oversubscription ratio was 4, 2, and 1, respectively. The client requests had a 10:1 GET-SET ratio with 128-byte key size and 2048-byte value size. Figure 12 shows the benefit of VB in improving the average, the 95^{th} percentile, 99^{th} percentile tail latencies ,and the throughput of the memcached server. As shown in the figure, thread oversubscription in the vanilla Linux did not inflict as much slowdown to memcached as did to many conventional parallel programs. Oversubscription incurred 6% increase in average latency and 5.6% drop in throughput. The results and experiments with other workloads in the Cloudsuite benchmarks (not shown due to the space limit), such as web serving, confirmed our findings that context switching does not drastically slowdown applications with relatively independent threads. However, as in many cloud workloads, memcached still employs traditional synchronization primitives based on futex and oversubscription led to 8x increase in the 95^{th} and 99^{th} percentile tail latency, as shown in Figure 12 (c and d). In contrast, VB effectively reduced the tail latency by 92% and 60%, respectively. Furthermore, VB was able to achieve close to the best performance as the number of core scaled.

4.3 Busy-waiting Synchronization

Micro-benchmarks. Spinning can cause devastating performance slowdown in oversubscribed scenarios because critical threads may be deprived of CPU cycles. To stress test our proposed busy-waiting detection, we designed a micro-benchmark with a multi-stage pipeline, with each stage assigned to a separate thread. Each thread spins on the completion of the previous stage before starting its own stage. As

Table 2: The *true positive* (TP) rate of BWD

Spinlocks	Alock-ls	CLH	Malth	MCS	Partitioned
# of Tries	55991	55859	55740	55718	55763
# of TPs	55860	55795	55676	55665	55686
Sensitivity(%)	99.76	99.88	99.86	99.9	99.86
Spinlocks	Pthread	Ticket	TTAS	CNA	AQS
# of Tries	55777	55769	55738	55785	55702
# of TPs	55699	55690	55664	55724	55635
Sensitivity(%)	99.86	99.85	99.86	99.89	99.88

Table 3: The *false positive* (FP) rate of BWD

App	# of Tries	# of FPs	Specificity(%)	FP overhead(%)
is	613136	3742	99.38	0.9
ep	3538136	2965	99.92	0
cg	4893039	27209	99.44	0
mg	1326357	3520	99.73	0.99
ft	3857623	572	99.99	0.26
sp	15075600	508	99.99	0.19
bt	10640650	9856	99.91	0
ua	12332790	2734	99.98	0.0043

such, the slowdown of one stage could cause cascading delays to the downstream stages. We studied 10 different spinlocks studied in [21]. Without oversubscription, threads ran on dedicated cores. We ran the micro-benchmarks in both containers and KVM virtual machines. While there are no spin detection mechanisms for containers [1] or native Linux, *pause loop exiting* (PLE) is able to detect spin loops implemented with the `NOP` instruction in VMs. The micro-benchmark was run on 8 cores with 8 threads or 32 threads. Figure 13 (a) and 13 (b) show that BWD can accurately identify busy-waiting in all spin algorithms and timely stop futile spinning. Across all benchmarks, BWD with 32 threads was able to achieve comparable performance to vanilla Linux with 8 threads. In contrast, PLE was not effective for any of the spin algorithms and performed similarly to the vanilla Linux.

User-customized spinning. In addition to the commonly used spinlocks, many applications implement customized busy-waiting algorithms. These algorithms are not only used as a means of synchronization but also as a building block of other functions, sometimes simply as a delay loop. Only spinning used as synchronization, which introduces inter-dependency among threads, is detrimental to performance under oversubscription. Figure 14 shows two examples of such programs, *lu* from NPB and *volrend* from SPLASH-2. We varied the number of threads from 8 to 32 and placed them on 8 cores. We made two observations. First, BWD was able to effectively bring the performance under oversubscription close to that without oversubscription, in both container and VM tests. PLE was not effective for user-customized spinning. Note that PLE is not applicable to the container case. Second, BWD inflicted some slowdown compared to the baseline and the slowdown worsened as the oversubscription ratio increased. The overhead is due to two reasons: 1) BWD may have false positives and mistakenly stop looping used in non-synchronization functions; 2) BWD detects spinning at fixed intervals (every 100 μs). As the number of threads increases, the aggregate amount of spinning would increase. Nevertheless, BWD

[1] Threads in containers are treated as ordinary threads in the host OS.

is significantly more efficient than the vanilla Linux and contains oversubscription overhead to an acceptable level.

True and false positive rates. To test BWD's true positive rate (sensitivity), we wrote a micro-benchmark with two threads placed on single core. Thread#1 continuously holds a spinlock while thread#2 repeatedly tries to acquire the spinlock. BWD's sensitivity is calculated as the number of detected spin loops divided by the number of lock acquisitions specified by the code. As shown in Table 2, BWD achieved close to 100% true positive rates for 10 different spinlocks. To test BWD's false positive rate (specificity), we chose 8 blocking-based benchmarks without any user or kernel-level spinning. Since the benchmarks contains no spinning, any detected spinning is a false positive. Table 3 shows that BWD's false positive rate was at most 0.61% across the 8 benchmarks. False positives occur when BWD encounters tight repeating loops with little data access. However, such loops are rare and BWD's mis-detection does not cause noticeable slowdowns, and the overall timer overhead is less than 3%.

4.4 Comparison with SHFLLOCKS

SHFLLOCKS [21] is a recently proposed lock design that decouples lock acquisition from a lock policy enforcement. It allows threads in the waiter queue to implement various locking policies and optimizations. An important feature of SHFLLOCKS is to allow waiter threads to adjust their lock policies, i.e., NUMA-awareness and efficient parking/wakeup strategies. SHFLLOCKS manages both the active and passive waiters in the same queue to decrease memory footprint, and enables lock stealing and shuffling to efficiently wake up waiters. This design improves fairness and throughput under oversubscription. We performed a comparison between SHFLLOCKS and our approach as well as the two spin-then-park algorithms evaluated in [21]. i.e., *Mutexee Locks* (Mutexee)[14], *MCS-TP lock* (MCS-TP)[17]. We replaced the pthreads primitives in slected benchmarks with the algorithms in the SHFLLOCKS library. Note that our approach requires no changes to application code and is compatible with pthreads.

Figure 15 shows the performance of five benchmarks with an oversubscription ratio of 4, i.e., 32 threads on 8 cores. The results indicate that spin-then-park algorithms still suffered drastic slowdowns when threads are oversubscribed while our approaches, VB and BWD, are up to 5.4x more efficient. The culprit was that these spin-then-park algorithms still rely on the futex interface for sleeping in the OS kernel, which may cause severe slowdowns under oversubscription. SHFLLOCKS performed even worse in this case. In an oversubscribed scenario, simultaneously waking up a large number of threads causes unnecessary migrations across cores. Not only does SHFLLOCKS have no optimizations for bulk wakeups but its NUMA-awareness may hurt performance as it always wakes up threads from the same socket, causing load fluctuations.

4.5 Limitation and Discussion

We have shown that oversubscription is a practical approach to exploiting CPU elasticity in the cloud. Many applications, mostly

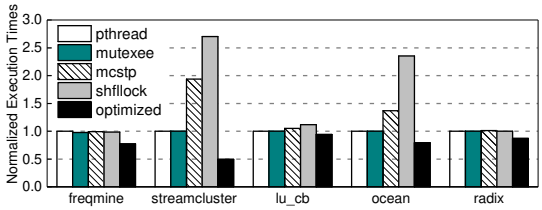


Figure 15: The performance due to shfllock, our approach, and other locks

embarrassingly-parallel workloads without much inter-thread synchronization, already run efficiently under oversubscription. We identified two issues that cause drastic slowdowns to synchronization-heavy applications when threads are oversubscribed: inefficient sleep/wakeup and wasteful spinning. We developed VB and BWD to effectively address these issues. Our results showed that VB and BWD together can effectively bring performance under oversubscription close to that without oversubscription. Nevertheless, there are limitations in our approach. We target applications that can scale with a fixed problem size (strong scaling) and our approach may not work well for workloads that have more work to do with more threads. *Fluidanimate* is one such application. The number of mutex locks in *fluidanimate* increases as more threads are provisioned, thereby suffering inevitable slowdown when oversubscribed.

5 RELATED WORK

Elastic computing has been studied to provide on-demand scalability as well as reducing the cost of leasing cloud resources [29]. However, most existing work on CPU elasticity focused on dynamically adjusting the number of CPUs allocated to virtual instances, such as VMs [11, 23] and containers [32]. It remains a challenge to adjust thread-level concurrency to utilize variable CPUs.

Adjusting thread-level concurrency Some recent work designed automatic parallelization techniques to manage the number of active threads based on different strategies [13, 19, 20, 34]. The optimal number of threads is determined either by system load [20], thread progress [8], synchronization overhead [25], or considerations on locality [13]. Arachne [34] is a userspace threading approach that used kernel threads as the proxy for dynamic core allocation. tScale [7] is a user-level lock-contention aware scheduler that manages thread count based on lock contention. However, these approaches require extensive changes to application source code or libraries to enable dynamic threading. In contrast, we focus on improving the efficiency of managing oversubscribed threads in the OS kernel.

Contention- and locality-aware lock design The inefficiencies we identified in thread oversubscription are due to synchronization. There has been work studying the efficiency and scalability of parallel program under contention [4, 6, 16, 35]. Gls [1] dynamically adapted locking algorithms under varying contention by monitoring the contention level. Kashyap et al. [22] introduced scalable NUMA-aware blocking primitives to handle both under- or oversubscribed scenario. Spinlocks are considered more harmful under high contention [37]. Li et al. [27] proposed a hardware thread spinning detection mechanism by tracking changes in CPU registers. Chakraborty et al. [9] detected spinning by checking the number of

unique stores executed in N committed instructions. Our findings suggest that L1 TLB misses and data cache misses together with branch addresses are reliable methods for busy-waiting detection. Results show that this method is effective for various spin implementations.

6 CONCLUSIONS

In this paper, we demonstrated that thread oversubscription can be made a practical approach to exploiting CPU elasticity. Contrary to traditional beliefs, provisioning more threads on a few cores does not necessarily lead to high overhead or dramatic performance slowdown. Through a systematic study of the direct and indirect costs of oversubscription as well as OS-level inefficiencies, we identified the culprits and addressed them via two new mechanisms, virtual blocking and busy-waiting detection. Results show that oversubscribed threads can be efficiently managed in the OS, incentivizing cloud users to provision more threads for future expandability.

7 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful feedback. This work is supported by National Key Research and Development Program under grant 2018YFB1003600, National Science Foundation of China under grants No.62032008 and 61872155. The corresponding authors are Song Wu and Jia Rao.

REFERENCES

[1] Jelena Antić, Georgios Chatzopoulos, Rachid Guerraoui, and Vasileios Trigonakis. 2016. Locking made easy. In *Proceedings of the International Middleware Conference (Middleware)*. 1–14.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *ACM SIGOPS operating systems review* 37, 5, 164–177.

[3] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 72–81.

[4] Hans-J Boehm. 2007. Reordering constraints for pthread-style locks. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 173–182.

[5] Carl Boettiger. 2015. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review* 49, 1 (2015), 71–79.

[6] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2012. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*. 119–130.

[7] Miao Cai, Shenming Liu, and Hao Huang. 2017. tScale: a contention-aware multithreaded framework for multicore multiprocessor systems. In *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS)*. 334–343.

[8] Gaurav Chadha, Scott Mahlke, and Satish Narayanasamy. 2012. When less is more (LIMO): controlled parallelism for improved efficiency. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. 141–150.

[9] Koushik Chakraborty, Philip M. Wells, and Gurindar S. Sohi. 2011. Supporting overcommitted virtual machines through hardware spin detection. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 23, 2 (2011), 353–366.

[10] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55.

[11] Wesam Dawoud, Ibrahim Takouna, and Christoph Meinel. 2011. Elastic vm for cloud resources provisioning optimization. In *Proceedings of the International Conference on Advances in Computing and Communications (ICACC)*. 431–445.

[12] Dave Dice. 2017. Malthusian locks. In *Proceedings of the European Conference on Computer Systems (Eurosys)*. 314–327.

[13] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach. 2016. Scalable task parallelism for numa: A uniform abstraction for coordinated scheduling and memory management. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*. 125–137.

[14] Babak Falsafi, Rachid Guerraoui, Javier Picorel, and Vasileios Trigonakis. 2016. Unlocking energy. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 393–406.

[15] Thomas Gleixner and Douglas Niehaus. 2006. Hrtimers and beyond: Transforming the linux time subsystems. In *Proceedings of the Linux Symposium*, Vol. 1. 333–346.

[16] Hugo Guiroux, Renaud Lachaize, and Vivien Quéma. 2016. Multicore Locks: The Case Is Not Closed Yet. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 649–662.

[17] Bijun He, William N. Scherer, and Michael L. Scott. 2005. Preemption adaptivity in time-published queue-based spin locks. In *Proceedings of the International Conference on High-Performance Computing (HIPC)*. 7–18.

[18] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. 2013. Elasticity in Cloud Computing: What It Is, and What It Is Not. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*. 23–27.

[19] Jialu Huang, Prakash Prabhu, Thomas B. Jablin, Soumyadeep Ghosh, Sotiris Apostolakis, Jae W. Lee, and David I. August. 2016. Speculatively exploiting cross-invocation parallelism. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*. 207–221.

[20] Ryan Johnson, Radu Stoica, Anastasia Ailamaki, and Todd Mowry. 2010. Decoupling contention management from scheduling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 117–128.

[21] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. 2019. Scalable and practical locking with shuffling. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. 586–599.

[22] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Scalable NUMA-aware blocking synchronization primitives. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 603–615.

[23] Ozgur Kilic, Spoorti Doddamani, Aprameya Bhat, Hardik Bagdi, and Kartik Gopalan. 2018. Overcoming Virtualization Overheads for Large-vCPU Virtual Machines. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 369–380.

[24] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. Kvm: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, Vol. 1. 225–230.

[25] Janghaeng Lee, Haicheng Wu, Madhumitha Ravichandran, and Nathan Clark. 2010. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, Vol. 38. 270–279.

[26] Chuanpeng Li, Chen Ding, and Kai Shen. 2007. Quantifying the Cost of Context Switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science (ExpCS)*. Article 2.

[27] Tong Li, Alvin R. Lebeck, and Daniel J. Sorin. 2006. Spin detection hardware for improved management of multithreaded systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 17, 6 (2006), 508–521.

[28] Tim Lindholm and Frank Yellin. 1997. Inside the Java virtual machine. *Unix Review* 15, 1 (1997), 7.

[29] Qixiao Liu and Zhibin Yu. 2018. The elasticity and plasticity in semi-containerized co-locating cloud workload: A view from Alibaba trace. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. 347–360.

[30] Jack Lo. 2005. VMware and CPU virtualization technology. *World Wide Web Wlectronic Publication* (2005).

[31] Victor Luchangco, Dan Nussbaum, and Nir Shavit. 2006. A hierarchical CLH queue lock. In *Proceedings of the European Conference on Parallel Processing (Euro-Par)*. 801–810.

[32] Jose Monsalve, Aaron Landwehr, and Michela Taufer. 2015. Dynamic cpu resource allocation in containerized cloud environments. In *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER)*. 535–536.

[33] NPB. 2019. *NAS Parallel Benchmarks*. https://www.nas.nasa.gov/.

[34] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: core-aware thread management. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 145–160.

[35] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. 2010. Analyzing lock contention in multithreaded applications. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Vol. 45. 269–280.

[36] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. 24–36.

[37] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. 2010. Ad Hoc Synchronization Considered Harmful. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 163–176.