User-level Threading: Have Your Cake and Eat It Too

MARTIN KARSTEN and SAMAN BARGHI, University of Waterloo, Canada

An important class of computer software, such as network servers, exhibits concurrency through many loosely coupled and potentially long-running communication sessions. For these applications, a long-standing open question is whether thread-per-session programming can deliver comparable performance to event-driven programming. This paper clearly demonstrates, for the first time, that it is possible to employ user-level threading for building thread-per-session applications without compromising functionality, efficiency, performance, or scalability. We present the design and implementation of a general-purpose, yet nimble, user-level M:N threading runtime that is built from scratch to accomplish these objectives. Its key components are efficient and effective load balancing and user-level I/O blocking. While no other runtime exists with comparable characteristics, an important fundamental finding of this work is that building this runtime does not require particularly intricate data structures or algorithms. The runtime is thus a straightforward existence proof for user-level threading without performance compromises and can serve as a reference platform for future research. It is evaluated in comparison to event-driven software, system-level threading, and several other user-level threading runtimes. An experimental evaluation is conducted using benchmark programs, as well as the popular Memcached application. We demonstrate that our user-level runtime outperforms other threading runtimes and enables thread-per-session programming at high levels of concurrency and hardware parallelism without sacrificing performance.

CCS Concepts: • **Software and its engineering → Concurrent programming structures**; **Multithreading**; **Software performance**.

Additional Key Words and Phrases: synchronous programming; user-level runtime; network server

ACM Reference Format:
Martin Karsten and Saman Barghi. 2020. User-level Threading: Have Your Cake and Eat It Too. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 1, Article 17 (March 2020), 30 pages. https://doi.org/10.1145/3379483

1 INTRODUCTION

Various network-based server applications, for example database or web servers, handle a multitude of stateful sessions, each comprised of multiple request/reply interactions driven by network communication. Because the number of concurrent sessions vastly exceeds the available hardware parallelism, sessions need to be suspended and resumed in software to mask the asynchronous I/O latency arising from network communication. An attractive programming model for these applications is *thread-per-session*, where each application session is represented by a software thread with an execution stack that provides automatic state capture (see also "Synchronous/Asynchronous Programming Models" in [10]).

Operating systems typically provide system threads with a certain level of convenience (e.g., thread-local storage) and protection (preemptive fair scheduling). *Blocking* I/O operations automatically suspend and resume a thread in response to I/O events. However, kernel-based system

Authors' address: Martin Karsten, mkarsten@uwaterloo.ca; Saman Barghi, sbarghi@uwaterloo.ca, University of Waterloo, Waterloo, Ontario, Canada.

2476-1249/2020/3-ART17 $15.00
https://doi.org/10.1145/3379483

threads are limited in number, due to complex scheduling, and have possibly high overhead, because creation, deletion, or blocking, involve in-kernel context switching. User-level threads are a light-weight alternative to handle a larger number of concurrent sessions, because of lower per-thread overhead.

The main alternative to the thread-per-session model is *event-driven* programming, where an application must explicitly manage the set of session states in response to I/O events reported by the operating system. This programming model is directly supported by typical operating system interfaces using the `kqueue` (*BSD) or `epoll` (Linux) system calls, which are improvements over the earlier POSIX interfaces `select` and `poll`. Programming libraries, such as libevent [2], libev [3], or libuv [4], provide convenient and portable interfaces for those system calls.

A user-level runtime uses the same operating system interfaces internally, but also provides a user-level blocking I/O interface to support the thread-per-session paradigm. A user-level scheduler decides which thread to execute next, based on I/O-readiness and other considerations. System-level blocking must be mitigated and managed by this scheduler and the user-level I/O subsystem. The *M:N threading* model describes a setup where many user-level threads (*M*) are transparently executed by multiple system threads (*N*) to facilitate automatic load balancing.

Other components in the technology stack, such as device communication and network processing in the operating system kernel and system libraries, also have a profound effect on application performance. One objective of this paper is isolating the performance implications of user-level threading from other system components. The other objective is demonstrating that user-level threading is possible at low effective overhead and thus without sacrificing performance. The paper's contributions are:[1]

- A scheduling scheme *Inverted Shared Ready-Stack* is proposed that efficiently schedules user threads on kernel threads using work-stealing, but also parks idle kernel threads.
- A corresponding user-level M:N threading runtime is presented that provides application concurrency with high efficiency, performance, and scalability – at a level not available with other existing user-level threading systems.
- A well-known server application, *Memcached*, has been transformed to the thread-per-session execution model. This permits an apples-to-apples performance comparison between event-driven and thread-based concurrency using the same production-grade application.
- We make no particular assumptions about hardware or the runtime setup of applications and try to avoid the need for opaque configuration parameters as much as possible.
- The experimental results confirm that our runtime provides consistently high performance – on par with the fastest event-driven designs.

The next section provides background information and reviews related work. In Section 3, a high-level overview of the runtime system is presented. This is followed by a discussion of the most important design specifics in Section 4. The evaluation is presented in Sections 5 & 6. The paper is wrapped up with a summary of our findings and a brief conclusion in Section 7.

2 BACKGROUND AND RELATED WORK

The original debate around thread- vs. event-driven programming was centered around the need to drive many concurrent client/server sessions in a single-CPU Internet server. This challenge has been termed the C10K problem [37], and has been restated recently as the C10M problem [27] for modern hardware. The main arguments against thread-per-session programming are the complexity of synchronization for non-deterministic (e.g., preemptive) scheduling, the resulting development cost, and the performance overhead associated with a thread runtime system [45]. The main

[1]Software and supplementary material available at `https://cs.uwaterloo.ca/~mkarsten/fred/`

counter-argument is the obfuscated non-linear control flow of event-driven applications [5, 61]. A good summary of the debate, including further references, can be found in Section 4.3 of Erb's thesis [21].

The debate faded without a definite resolution. Contemporary software systems seem to favor event-driven programming, but especially the advent of Go [34] and its usage in cloud software components has renewed interest in multi-threading as a programming technique [24]. A key component for many concurrent applications is the I/O interaction pattern. Without sufficient thread concurrency, event-based systems must use non-blocking and/or asynchronous system call interfaces [20]. Non-trivial event-driven programming might lead to a phenomenon colloquially termed *callback hell* [32]. In contrast, multi-threading with I/O blocking permits a synchronous programming style for per-session processing. The complexity of continuations is hidden by automatic state capture provided by stack-based threading [10]. A proper user-level threading runtime provides a user-level-blocking I/O interface. In this case, the efficiency of the corresponding user-level I/O subsystem is crucial for the performance of I/O-heavy applications.

A particular line of investigation in this context has focused on web servers and has produced whole-system evaluations of thread- vs. event-driven concurrency. The Staged Event-Driven Architecture (SEDA) proposes a hybrid architecture to decompose event handling into multiple stages, executed by thread pools, which are subject to dynamic resource control [64]. The main objective is reducing the complexity of event-driven systems without compromising performance. The Capriccio user-level threading runtime, tested with the Knot web server, explores dynamic stack allocation and resource-aware thread scheduling, among other details, to achieve good resource efficiency in a thread-per-session application [62]. Pariag et al. present a systematic comparison of thread-based vs. event-driven web server architectures at the time, along with a pipelined design that generalizes SEDA [46]. Subsequent work has extended this investigation to include small-scale multi-core platforms [31]. In all cases, results are reported for custom web servers and read-only workloads, not shared mutable data. The most important finding from this work is that attention to detail matters. Web server performance does not only depend on the programming paradigm, but also on configuration decisions related to device communication, disk I/O, network stack, and system calls. However, our work is focused on software efficiency depending on the choice of runtime paradigm. For this reason, we exclude kernel extensions [7, 59], advanced locking strategies [18, 29, 41, 53], and lock-free algorithms [13, 28] from direct consideration.

Most earlier work, such as SEDA [64] and Capriccio [62] has studied single-core processor hardware and thus has only limited applicability on modern hardware platforms. The prevalence of multi-core systems for high-performance server applications makes an important part of the original argument in favor of event-driven programming moot. Multiple cores can only be utilized via system threads, while parallel execution and/or preemptive kernel scheduling require synchronization when accessing shared mutable data or resources. An event-driven application with shared mutable data can run a sequential event-loop per system thread, but cannot avoid the complexity of thread synchronization. For example, Memcached [42] is a production-grade high-performance key/value store that uses this design. Others have investigated highly tuned variations of key/value stores [39]. In contrast, we have made as few modifications to the original Memcached code as possible – to emphasize the effects of different underlying runtime systems.

Several user-level threading systems implement the N:1 threading model, where multiple user-level threads are executed by a single system thread, similar to Capriccio [62]. Examples include Facebook's folly::fibers [23], GNU Pth [49], State Threads [56], or Windows Fibers [66]. In these systems, each system thread typically runs a separate I/O event handling loop. The association of user-level threads to system threads must be managed by the application, for example to facilitate load balancing across system threads. In contrast, an M:N threading runtime transparently executes many user-level threads (M) on multiple system threads (N), i.e., scheduling and I/O event handling are logically centralized.

A number of established programming languages provide M:N threading via their runtime environments and have specific objectives beyond providing multi-threading. μC++ [14] is focused on embedding concurrency into an object-oriented programming language. Go [26] supports concurrent programming following the CSP model [34] and, most importantly, also provides managed memory. Erlang [22] is a functional programming language with a managed runtime system, where concurrency follows the actor model [33] and is implemented via user-level threading. The runtime systems of these programming languages represent specific design trade-offs between the desired high-level functionality and performance. Quasar [51] adds lightweight threading to Java. Both Erlang and Java execute bytecode in a virtual machine, which is currently beyond the scope of our work. These additional features make it difficult to isolate the performance impact of multi-threading and to perform an apples-to-apples comparison using an existing event-driven application. However, Go is included in the first part of the experimental evaluation, because it is compiled to machine code and provides user-level threading.

Several user-level runtime systems providing M:N threading exist in the realm of unmanaged C/C++ execution: Boost Fiber [12], Libfiber [40], Mordor [43], and Qthreads [65] are included in our evaluation – demonstrating that existing M:N runtimes do not realize the full potential of user-level threading. Other runtime systems [9, 44] are not competitive and thus not included.

A recent notable study warrants special attention: Independent of our work, Arachne has been proposed as a high-performance user-level threading runtime [50] – and has been evaluated using Memcached, as well. The key contribution of Arachne is illustrating the benefits of user-level load balancing in reducing tail latency under certain circumstances. However, Arachne is not a complete user-level threading runtime and both its system design and evaluation methodology have shortcomings. For example, the number of active threads is limited by the thread dispatch scheme to 56 per core. The most important design deficiency is the lack of user-level I/O blocking and the corresponding inability to support the thread-per-session programming model. Instead, a thread-per-request model is implemented. This works for Memcached's stateless requests, but cannot support a synchronous control flow for stateful application sessions. In addition, the experiments in Sections 5 & 6 demonstrate Arachne's performance limitations. The (potential) latency benefits of user-level load balancing are provided by our runtime system as well (see Section 6.3 and Appendix B), but without the inherent compromises made in the design of Arachne.

An separate formal model for highly concurrent systems is *actor* programming. In some sense, the actor model provides a formal framework for event-driven programming. The original model [33] mandates message passing and no I/O blocking. In practice, actor runtimes internally use event-loops in combination with system threads [6, 16, 30, 47] and/or map actors to user-level threads [22, 48], so our findings are applicable in this area, as well.

3 RUNTIME SYSTEM OVERVIEW

An important objective for our runtime system is flexibility and usability. In particular, we make no fundamental assumptions about runtime scenarios. Applications are linked with a library, but can be executed in any environment that can run regular C/C++ programs. The runtime library itself is implemented in C++, but also provides a C API that mimics the *pthreads* API. It can be built and used with GCC or Clang on Linux or FreeBSD on x86-64 hardware platforms, but there is no inherent obstacle for porting it to other platforms. However, only the Linux/GCC version is evaluated in this paper. Interactive debugging is supported as a Gdb extension.

The runtime provides execution contexts that are independently suspended and resumed with automatic state capture backed by a stack. A context resembles a traditional thread, but without preemption. This is often called a “fiber”. However, the large number of existing runtime systems (see Section 2) results in overloaded terminology. Therefore, in this paper, we denote an execution context in our runtime as *fred* (rhymes with “thread”). The fred runtime provides M:N threading for fred execution, that is, many user-level freds (M) are transparently executed by several system threads (N).

The runtime system supports both round-robin and load-based placement of new freds. Its scheduler is non-preemptive to limit complexity and to avoid reentrancy problems with third-party libraries. Because all freds belong to a single application, developers have other means to ensure responsiveness without relying on preemption as a resource arbiter. A scheduling domain, called *cluster*, realizes the M:N threading model by transparently executing freds on a set of *processors*, which are an abstraction for system threads and ultimately CPU cores. Applications employing many cores across multiple NUMA nodes can be split into multiple clusters to work around inherent scalability bottlenecks. Shared memory, synchronization, and fred migration are fully available across clusters.

The programming interface is straightforward. An application can dynamically create, update, and arrange fred, processor, and cluster objects for a desired execution layout. A typical setup uses one processor object per CPU core and one fred per application session. User-level blocking synchronization mechanisms, such as mutex, condition variable, and semaphore, are provided with timeout support. By default, freds resume execution on the same system thread on which they were suspended. However, the runtime supports explicit fred migration and the scheduler performs automatic load balancing via transient work-stealing.

User-level I/O blocking is provided by an I/O subsystem and accessed through three types of interfaces: 1) Most I/O operations are encapsulated by a generic wrapper that facilitates user-level blocking. 2) Specific I/O operations, for example those that need to register a new file descriptor (FD) with the I/O subsystem, are encapsulated via specific wrapper functions. 3) In cases where I/O operations are not supported by OS-level event notifications, such as disk I/O on Linux, they can be invoked via another generic wrapper. This transparently executes the corresponding operation on a dedicated system thread (of a pool), so that the I/O wait time does not affect the execution of other freds.

The runtime system supports resizable stacks using the *split stack* feature provided by GCC [35] and Clang [55]. Split stacks are very lightweight and only add a few instructions to each subroutine. Preliminary experiments indicate no discernible performance impact. However, a detailed investigation is beyond the scope of this paper.

4 DESIGN DETAILS

The canonical objectives for any computer system are high performance (in terms of high throughput and low latency) and good resource efficiency. A runtime system must facilitate these objectives for applications in a scalable fashion at increasing levels of hardware parallelism. When translating these objectives into actual mechanisms, however, conflicts arise. For example, busy-looping idle processors is good for latency and throughput, but parking system threads during times without work (*idle-sleep*) is crucial for resource efficiency. As another example, tight control via blocking locks is good for resource efficiency, but hurts scalability. The design of central components must address these trade-offs.

4.1 Scheduling

The scheduler matches executable freds with available processors and sits at the crux of the problem of providing high performance and good resource utilization. It needs to be effective in keeping processors busy, if work is available. Uneven workloads must be balanced across available processors to achieve high performance. At the same time, it also needs to be efficient and scalable. As in other user-level runtimes, there is no expectation of service guarantees for specific freds. Within a single application, it is less important when a fred is executed, as long as useful work is performed efficiently.

4.1.1 Ready-Queue(s). The simplest possible design uses a central shared ready-queue that is served by all processors (used, for example, by μC++). This trivially provides excellent load balancing, but the resulting contention often turns the shared queue into a bottleneck and inhibits scalability. Furthermore, a shared ready-queue does not support *affinity*, by which a fred repeatedly executes on the same core to potentially benefit from a warm cache.

At the other end of the spectrum is a design with processor-local ready-queues, which, however, requires other means for load balancing, such as work stealing [11]. The combination of work stealing and idle-sleep leads to a potential race condition, by which the last processor going to sleep might miss a wake-up signal. Other M:N runtimes that implement work stealing solve this problem at varying degrees of complexity, but ultimately defer to a busy-loop to address any loose ends after mitigating the race condition. Boost Fiber and Qthreads simply busy-loop all system threads. This approach is a suitable strategy for an HPC application that usually keeps a dedicated set of cores busy. However, it is not an acceptable scheduling regime for applications in a shared computing environment, especially when also considering power conservation. Libfiber busy-loops all system threads, but injects sleep times to reduce CPU utilization. Arachne busy-loops in principle, but also dynamically estimates an effective load level based on idle and busy times to decide how many system threads are needed to handle the current workload. Both Arachne and Go ultimately defer to a single representative system thread. That final system thread busy-loops in Arachne, while Go injects sleep times to reduce CPU utilization.

The fred runtime uses a hybrid solution with both shared and local queues as illustrated in Figure 1. A processor searches for an executable fred first in its local ready-queue, then in a cluster-global staging-queue, and then by stealing. All processors in a cluster are organized into a ring. Work-stealing searches for a victim along that ring, starting at the thief processor. We have found this simple scheme sufficient for our purposes, but future work might refine or enhance it. If a fred is taken from the staging-queue, it can be moved to a particular processor. This mechanism is used for load-based placement and can also be used for application-level work sharing. If an application detects a load imbalance, freds can be moved from highly loaded processors to the staging-queue. Those freds are automatically picked up when lightly loaded processors run out of work. The staging-queue is also used to facilitate shared *background* work. A fred can be marked as 'do-not-move' and is then repeatedly placed in the staging-queue. This can be used for accepting new work only if there is capacity.

4.1.2 Idle Sleep. The key component of the runtime scheduler is a new scheme termed *Inverted Shared Ready-Stack* to avoid busy-looping. It logically operates as a semaphore, but is implemented like a benaphore [54] for efficiency. It is illustrated in Figure 1 and pseudo-code is provided in Algorithm 1. An atomic variable ($fred\_counter$) counts the total number of ready freds in a cluster or, when negative, denotes the number of processors waiting for work. Each fred arrival increments the counter, akin to a `V()` operation (Line 8). Before searching for an available fred, each processor decrements the counter, equivalent to `P()`, to reserve a fred in one of the queues (Line 23). When the system is backlogged (freds available, no processor waiting), the overhead of this scheme is a single atomic arithmetic instruction per enqueue or dequeue operation. If a processor finds the system empty, it is placed on a stack and parked using OS-level blocking (Lines 28-30). If the system is completely idle, wake-up (including timers) is delegated to the I/O subsystem, presented in the next section. If a fred arrives while processors are parked, a processor is removed from the stack

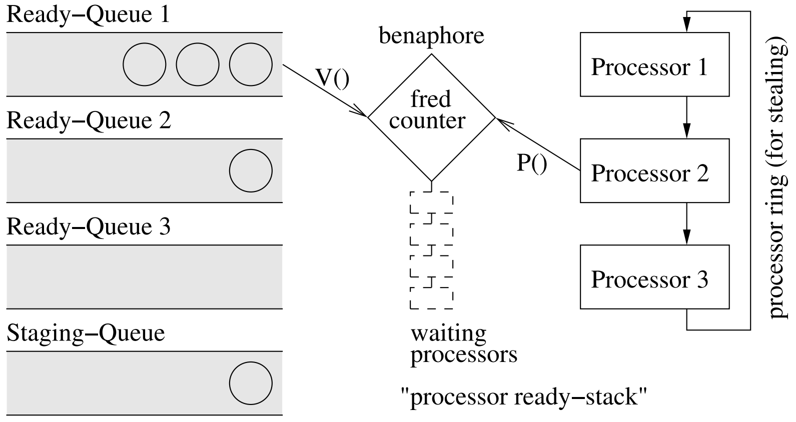


Fig. 1. Fred Scheduler

and resumed (Lines 16-18). The arriving fred is directly handed over to the resumed processor, instead of being placed in a ready-queue.

A potential race condition arises between the respective atomic operation and locking (Lines 8/11 vs. 23/26), if a processor has already decremented the counter, but is not yet added to the processor stack. This race condition is handled through a helper data structure $temp\_fred\_list$ that temporarily stores freds for such 'slow' processors (Line 13) and facilitates the fred handover (Line 32). A similar race condition applies to releasing the shared scheduler lock and suspending/resuming the processor (Lines 17/18 vs. 29/30). This is handled, similarly, by using dedicated OS-level semaphores for suspend and resume, in combination with fred handover. Using a dedicated semaphore per processor is ultimately more efficient than extending the holding time of the shared scheduler lock. Finally, a third race condition exists between counting a fred and the corresponding queue operation (Lines 8/9 vs. 23/24). This is addressed by performing the fred search operation in a continuous loop (Line 24). The search is guaranteed to succeed and does not take long, because a fred is effectively reserved at this point and will appear in one of the queues very shortly.

Where normally a shared ready-queue would store freds, this data structure is *inverted* and holds processors. Instead of a queue, parked processors form a *stack* to improve the chances that CPU cores corresponding to longer-idle processors can reach deeper idle states. As the key benefit, the overhead of locking a shared mutable data structure and OS-level blocking only accrues, if the system is not fully loaded. Thereby, the inverted shared ready-stack provides accurate, yet efficient, accounting of executable freds – or parked processors – in a cluster. It thus trivially addresses the potential idle-sleep race condition arising from work stealing.

4.2 User-level I/O Blocking

The runtime system provides a user-level blocking interface for I/O operations and timers through a subsystem that internally uses non-blocking system calls and event polling. Socket/file descriptors (FD) typically have global scope across system threads in a process and POSIX mandates compact allocation at lowest possible numbers [36]. Thus, the corresponding user-level synchronization objects are stored in a global vector, indexed by the FD value. An extension of this model (*event scope*) is presented in Section 4.2.3. Synchronization is provided separately for the input and output direction of each FD. Each side is comprised of a user-level lock, used to serialize access from

Algorithm 1 Inverted Shared Ready-Stack

```
function Init( )
    $lock \leftarrow$ open
    $proc\_stack \leftarrow$ empty
    $temp\_fred\_list \leftarrow$ empty
    $fred\_counter \leftarrow 0$
end function
function EnqueueFred($f$)                                    ▷ fred $f$ to be made ready
    if atomic_add_fetch($fred\_counter$, 1) $> 0$ then
        add $f$ to local ready queue or staging queue
    else
        $lock$.acquire()
        if $proc\_stack$.empty() then
            $temp\_fred\_list$.push_back($f$)
            $lock$.release()
        else
            $p \leftarrow proc\_stack$.pop()
            $lock$.release()
            $p$.resume($f$)                                  ▷ hand over $f$
        end if
    end if
end function
function DequeueFred($p$)                                    ▷ procssor $p$ looking for work
    if atomic_sub_fetch($fred\_counter$, 1) $\geq 0$ then
        loop: get $f$ from local queue, staging, or steal
    else
        $lock$.acquire()
        if $temp\_fred\_list$.empty() then
            $proc\_stack$.push($p$)                          ▷ use list as stack
            $lock$.release()
            $f \leftarrow p$.suspend()                       ▷ hand over $f$
        else
            $f \leftarrow temp\_fred\_list$.pop_front()
            $lock$.release()
        end if
    end if
    return $f$
end function
```

multiple freds, and a user-level semaphore to signal I/O readiness. A *poller* object manages a subset of all FDs as the *interest set* and interacts with OS-level event mechanisms (explained next) to determine the readiness of FDs for input and/or output, as illustrated in Figure 2. In principle, the association between FD, poller, and cluster can be arbitrary within the same event scope. However, one typical layout uses one poller per cluster and registers each FD with the corresponding interest set.

Generally, I/O operations are executed as non-blocking system calls at the OS level. If an operation is user-level blocking and the non-blocking OS-level attempt cannot be completed, the operation

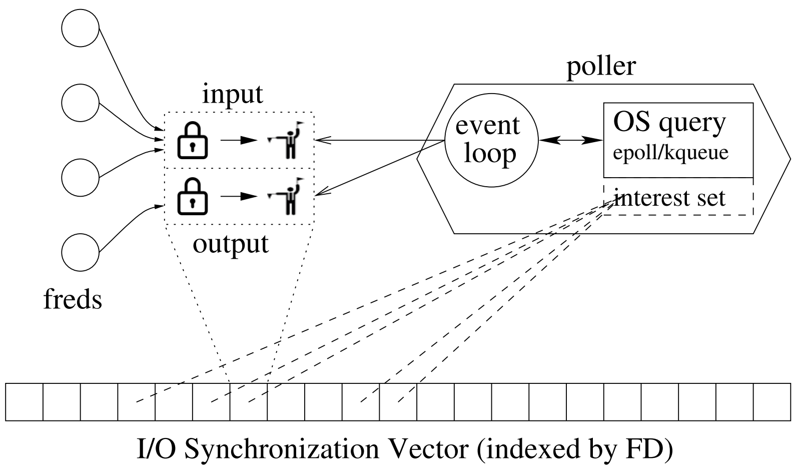


Fig. 2. I/O Blocking [1]

is synchronized with the poller and retried after the poller signals I/O readiness. This scheme is necessary to use the desired interaction pattern with event system calls (see next section). Two design details have been evaluated by preliminary experiments (not shown here) and have been found to improve performance: 1) Before attempting an input operation, a fred yields execution once. This improves the probability that the initial non-blocking OS-level attempt succeeds without resorting to synchronization. It also adds a measure of fairness to applications, because a busy session cannot hog resources. 2) An FD is only lazily added to the interest set, if and when a non-blocking operation attempt ever fails. This reduces registration and notification overhead, especially for short-lived connections.

4.2.1 Event Handling. The status of FDs is queried using the `epoll` event notification facility on Linux and `kqueue` on FreeBSD. While there are libraries providing a uniform interface across diverse OS mechanisms [2–4], a direct implementation ensures the greatest clarity and performance. In principle, Linux and FreeBSD provide an *edge-triggered* interface that delivers a single notification when the I/O readiness status of an FD changes. However, both share a peculiar behavior for edge-triggered notifications – most likely a result of those mechanisms having been added to established I/O subsystems: Notifications are not only generated when the status of a buffer actually transitions between empty, non-empty, and full, but the readiness of an FD is updated in the kernel after each input or output activity. Thus, any I/O activity for an FD potentially results in a new notification, delivered during the next call to `epoll_wait` respectively `kevent`. For example, multiple arrivals to a non-empty socket queue cause multiple "edge-triggered" events. In other words, this scheme only suppresses notifications, if there is no I/O activity at all for a particular FD between subsequent event queries. This behavior does not affect an event loop that synchronously alternates between event notifications and performing all resulting work. However, it can lead to *spurious notifications*, if an event loop is executed concurrently. Therefore, the frequency of OS-level queries presents a trade-off between efficiency and latency: more frequent queries reduce the notification latency, but increase the overhead caused by spurious notifications. The I/O subsystem in the fred runtime mitigates the effects of spurious notifications, for example by using binary semaphores that ignore repeated notifications.

4.2.2 Event Notification Strategy. Event handling for M:N threading operates differently from event-based systems. In a typical event-driven application, multiple FDs are associated with a specific system thread and that system thread synchronously alternates between receiving events and performing work. In contrast, an M:N threading runtime provides transparency and load-balancing across system threads. Because any user-level thread can access any file descriptor while executing on any system thread, there is no natural synchronous execution pattern for event notification queries. This presents a critical design challenge for the efficiency and performance of a user-level runtime system. The per-cluster poller in the fred runtime queries the "edge-triggered" OS interface and signals I/O readiness to interested freds using user-level synchronization. We have identified and implemented three variants for executing the poller, all of which integrate well with the idle-sleep property of the scheduler discussed in the previous section.

The simplest choice is a dedicated system thread that repeatedly makes blocking system calls to query for events. With this scheme (*thread poller*), the query frequency is determined by kernel-level scheduling, which means a lack of control for the user-level runtime and potentially many spurious notifications. Due to limited space, we only summarize the performance observations: A dedicated poller system thread provides reasonable performance, but is consistently outperformed by the other schemes.

A better design integrates the poller with scheduling, so that the runtime system has control over the query frequency. Other runtimes with user-level I/O blocking, such as Go or Libfiber, perform non-blocking queries when any system thread runs out of regular work. In Libfiber, these queries are part of the scheduling busy loop and also inject small amounts of sleep time. However, they are not coordinated and thus block each other via kernel locks, which leads to very poor overall performance (cf. Section 5.2).

The default scheme of the fred runtime (*fred poller*) executes the event query loop as a dedicated *background* fred (see Section 4.1) that is executed under runtime system control only when a system thread runs out of local work, which reduces spurious notifications. Coordination among multiple system threads happens automatically via fred scheduling. The fred poller makes several non-blocking attempts to query the event-FD for notifications (followed by yielding), but ultimately resorts to user-level blocking. This is facilitated by a master *thread poller* that uses `epoll` or `kqueue` hierarchically for the event-FD, but is only rarely active.

This scheme can be extended to a configurable number of poller freds per cluster. The complete set of FDs is partitioned (round robin) into multiple interest sets and each poller fred queries a particular interest set. With sufficient application dynamics, this leads to a pseudo-random distribution of FDs to interest sets, but this is not guaranteed. The advantages of multiple poller freds are given by the correspondingly smaller interest sets that are processed by each query and the potential for executing parallel queries. The primary drawback is possibly higher latency, because the I/O readiness of a particular FD might only be known after several queries. An extreme version of this scheme is termed *direct poller* and mimics the event handling behavior of event-based systems. One dedicated poller fred is created per processor and FDs are assigned to interest sets based on the executing processor.

4.2.3 Extension: Event Scope. Linux supports multi-threading with shared memory, but separate FD tables [58]. I/O-heavy applications that serve a large number of short-lived connections can benefit from improved kernel-level I/O scalability with this execution layout. In particular, the overhead of the `accept` system call is directly related to the size of the FD table. Separate kernel-level FD tables are represented in the fred runtime by the *event scope* concept. Each event scope contains a separate I/O subsystem and can hold multiple clusters. Multiple event scopes can co-exist in a process and still benefit from shared memory, user-level synchronization, and limited fred migration. However, their I/O subsystems are separate and handled through separate FD tables in the kernel.

4.3 Data Structures

We try to avoid reinventing the wheel and have carefully chosen known and simple data structures as much as possible. Our objective is establishing the validity of the overall design first, without muddying the waters with low-level optimizations that might or might not have a tangible impact on the performance bottom line.

Most containers are implemented as intrusive data structures, i.e., linkage fields are co-located with objects to reduce memory allocation overhead. All logical queues from Figure 1 are implemented as arrays of queues to support fixed static priorities. The default queue data structure is the lock-free Nemesis queue [15]. This is a *multi-producer single-consumer* queue, i.e., supports concurrent accesses from multiple producers and a single consumer. Thus, a lock is needed to coordinate concurrent dequeue operations from multiple consumers during work-stealing. Two alternatives, another single-consumer lock-free queue [19] and a traditional lock-protected queue, are also implemented and have been tested. The particular choice of queue data structure does not have a substantial impact on performance. Traversal of the processor ring in a cluster is safe from concurrent insert operations and therefore, does not require additional synchronization. System-level blocking operations are preceded by short spins of non-blocking attempts to avoid suspending and resuming system threads when possible. This applies to internal locks, event queries, and idle-sleep suspend.

5 BENCHMARKS

Micro-benchmarks are used to assess the fundamental scalability and efficiency of scheduling and user-level I/O blocking. The results demonstrate that the fred runtime provides competitive or superior performance compared to all other user-level threading systems and is on par with event-driven I/O handling. We do not directly investigate the performance of primitive operations, such as thread creation or yielding. In a realistic thread-per-session application the cost of accepting a new network connection dominates the cost of user-level session/thread creation, as shown by the Connection Churn experiment in Section 5.2.3. On the other hand, the cost of yielding is dominated by the indirect costs of cache evictions and difficult to determine with benchmark tests.

All experiments reported in this section are performed on a 4-socket, 64-core AMD Opteron 6380 with 512 GB RAM, but have also been repeated on a 4-socket, 32-core, 2xHT Intel Xeon E5-4610v2 with 256 GB RAM – to the extent possible. Only the AMD results are presented here, because some experiments on the Intel machine have to be scaled down to avoid using hyper-threaded execution. However, the results are generally consistent on both platforms. The operating system is Ubuntu Linux 18.04 running kernel version 4.15.0-47-generic with GCC 7.3.0. For all experiments, the specified set of cores is provisioned using the `taskset` utility. All data points show the average of 20 runs. Because the standard deviations are generally not significant, error bars would only distort the graphs and are not shown. In addition to event-based implementations and other user-level threading systems, we also typically compare to a benchmark or application version that directly uses system threads for concurrency. We label this variant 'Pthread'.

5.1 Scheduling

The scheduling benchmark is a program that implements a simple work model. N concurrent loops execute the following sequence in each loop iteration: private work w, then randomly acquiring one of l locks, then executing critical section work c, followed by releasing the lock. Both work

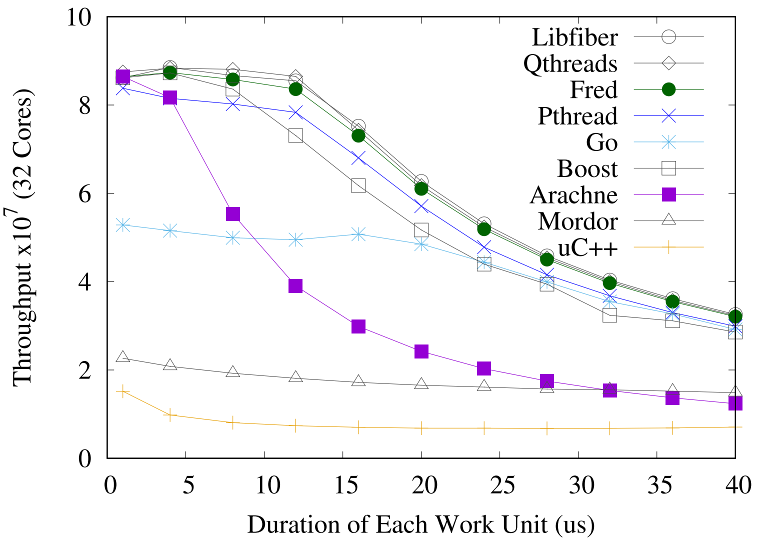


Fig. 3. Performance (1024 loops, 32 locks)

units can be calibrated to a specific time period. For simplicity, the experiments reported here use the same setting for private and critical-section work.

5.1.1 Scalability. Most multi-threading runtime systems perform well when the amount of work between synchronization and scheduling points is large. The first experiment is designed to investigate scalability by determining the amount of work that is necessary for runtime systems to provide good scalability at non-trivial core and concurrency counts. The benchmark program uses 32 cores and executes 1024 concurrent loops. It is configured with 32 locks, which causes some random contention and synchronization overhead, but not much. The total running time is fixed at 10 seconds and the work unit duration is varied. The throughput performance is shown in Figure 3. Go, Arachne, Mordor, and μC++ show noticeable weakness, although Go achieves similar throughput as the other runtimes at longer work units. To illustrate scalability independent of performance, Figure 4 shows each 32-core throughput normalized by the respective throughput when only a single loop (on a single core) is executed. Because of inherent overheads and contention, the normalized relative throughput never reaches 32, which would indicate perfect scalability. The best runtimes, Libfiber, Qthreads, and Fred, scale to 25X in this experiment. Pthread, Go, and Boost follow reasonably closely, while Arachne and Mordor show mediocre scalability. μC++ has the poorest scalability of the tested runtimes. However, additional experiments (not shown here) confirm that μC++'s scaling curve starts rising at 55 - 70 μs for both work units, beyond which it scales close to the top runtimes.

5.1.2 Efficiency. The previous experiment indicates that most runtime systems show reasonable scalability when each work unit is set to 20μs. The next experiment uses that setting to investigate the runtime overhead for locking and scheduling. The number of cores is varied and the number of concurrent loops is set to the square of the number of cores. The measured cost of each loop iteration is comprised of one unit of private work and one unit of critical-section work, plus synchronization overhead. Thus, any value in excess of $2 \cdot 20\mu s = 40\mu s$ represents runtime overhead. Since overhead

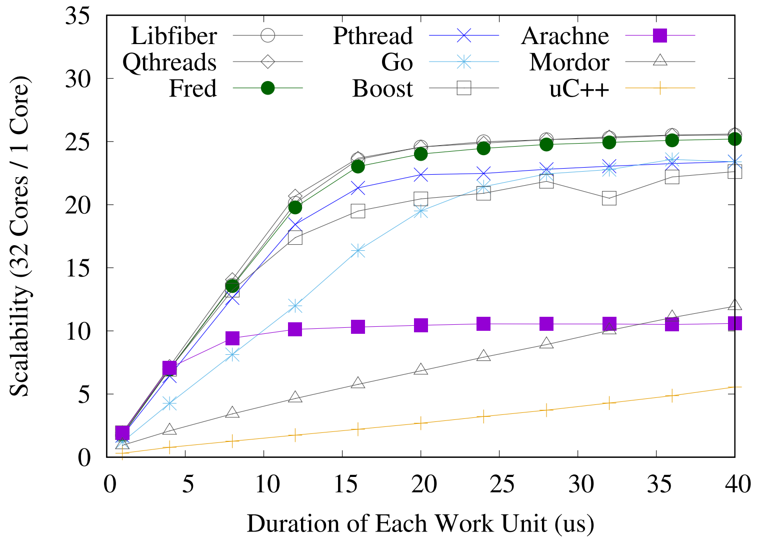


Fig. 4. Scalability (1024 loops, 32 locks)

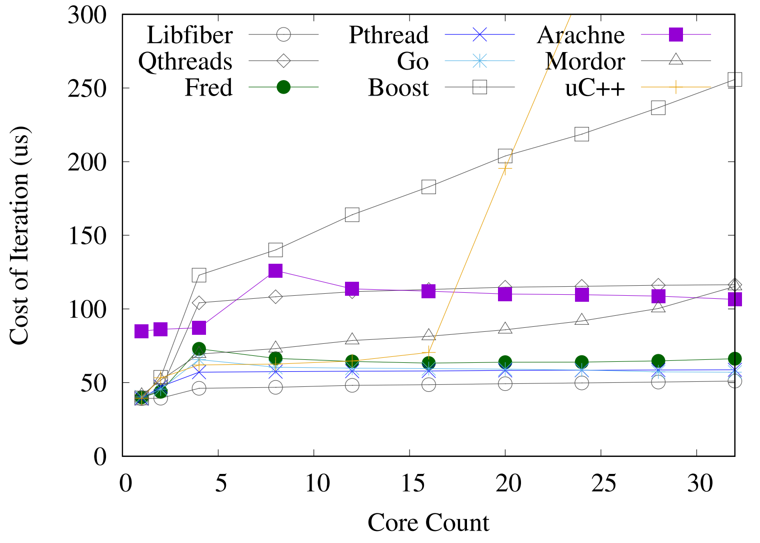


Fig. 5. Efficiency (N^2 loops, $N/4$ locks, 2x 20μs work)

becomes more visible with higher contention, the number of locks is set to $N/4$ for N cores, which ensures a sufficient level of contention. The results are shown in Figure 5.

μC++ shows good efficiency until crossing the NUMA boundary at 16 cores in this experiment, at which point efficiency becomes very poor. Boost also shows very poor efficiency at increased levels of hardware parallelism and concurrency. In both cases, this points to increasing overheads for mutex locking. Boost has been run with its default work-sharing scheduler, since the work-stealing scheduler would occasionally crash during this experiment. We suspect a bug, but have not investigated further at this time. Manual inspection shows that the best possible efficiency of Boost would be similar to Qthreads. Qthreads shows good scalability in the previous experiment, but poor efficiency in this one. As explained in Section 4.1, both Boost and Qthreads continuously spin all system threads while seeking work, which leads to poor resource efficiency. Arachne's efficiency is shown for the spin-lock variant. Because of spinning, the core arbiter has no opportunity to park system threads, which also leads to a situation where all cores continuously spin and thus poor efficiency. However, Arachne's blocking lock has much poorer scalability (not shown) and results in even worse efficiency (not shown). Mordor's efficiency varies with the level of hardware parallelism. Its limited efficiency fits the mediocre scalability shown in the previous experiment. Pthread, Fred, Go, and Libfiber provide effective blocking and show similar efficiency. Libfiber's efficiency is slightly, but noticeably, better than the rest. This is due to its simplistic scheduling that is very efficient, but cannot support proper I/O handling (see next section).

5.2 User-level I/O Blocking

Arachne, Boost, and Qthreads do not support user-level I/O blocking at all. Mordor claims I/O support, but in the absence of publicly available documentation, we have been unable to include it in this experiment. Given its limited scalability and efficiency as demonstrated by the previous experiment, we argue that this is not a big loss. Libfiber also claims to support user-level I/O blocking, but its simplistic approach to scheduling, as outlined in Section 4.1, results in multiple `epoll_wait` system calls fighting for kernel locks. The resulting serialization dramatically reduces I/O throughput and we have been unable to create and sustain much more than 100 concurrently active connections – even with a trivial test setup. This makes any further investigation moot.

Overall, we conclude that Boost, Libfiber, Mordor, and Qthreads are not competitive for the scenarios studied here and exclude them from further consideration. Because it does not support user-level I/O blocking, Arachne is excluded from I/O benchmarks.

The objective for these benchmark experiments is assessing the scalability and efficiency of user-level I/O blocking for those runtimes that support it. The experiment is executed locally on the multi-socket computer partitioned to run both client and server with communication across the loopback software interface. While there might be certain side effects of client and server sharing the same machine, this setup on the other hand limits the influence of hardware specifics. For example, it eliminates performance-masking latency effects, such as those investigated in Appendix B. The aim is to compare the inherent software efficiency and performance potential of different runtime systems for concurrent applications.

The benchmark is a variation of the *plaintext* benchmark from the TechEmpower benchmark suite [57]. A web server receives simple and short HTTP requests from clients and responds to each request with a plain "Hello, World!" HTTP response. This test stresses the I/O subsystem without performing much other work. Hence, this experiment represents a worst-case scenario for a user-level runtime by exposing the overhead from user-level I/O blocking without any mitigating factors – to the extent possible. A minimal custom thread-per-session web server that can use either system threads, our fred runtime, or μC++, is compared to a minimal Go web server that is derived from the publicly available TechEmpower benchmark code and uses the Fasthttp [60] package.

In addition to these multi-threaded web servers, we compare to a best-in-class event-based system: The ULib web server [17] consistently performs at or near the top of the TechEmpower benchmark, even when compared to small custom web servers. It uses a sophisticated and highly-tuned N-copy process architecture that does not permit any migration after connection establishment. The server socket is cloned across all processes using the `SO_REUSEPORT` socket option. In contrast, the Go server and our test server use a simple and straightforward architecture with a single dedicated 'acceptor' thread launching a new thread for each incoming connections.

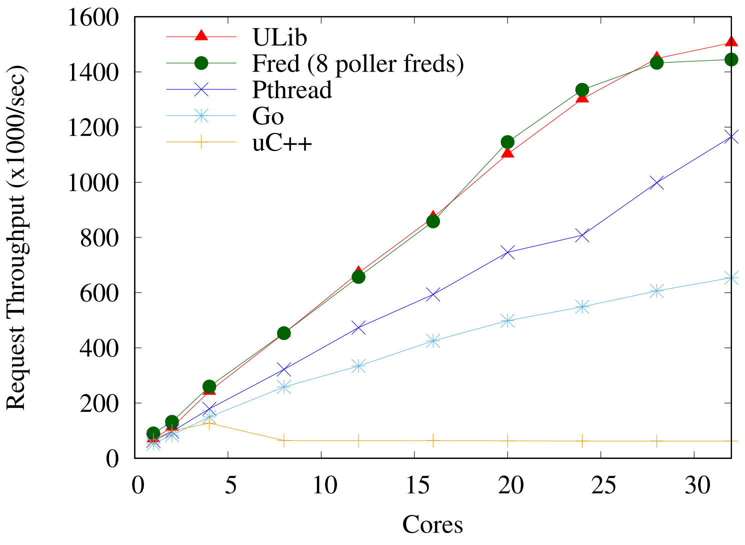


Fig. 6. Web Server (15K connections)

5.2.1 Core Scalability. The 'wrk' benchmark program [25] creates 15,000 connections and then exchanges requests and responses as fast as possible, using 32 out of the available 64 cores, for 60 seconds. Multiple connections provide a means for latency-masking concurrency in each client thread and the experiment does not use additional HTTP pipelining. The throughput is measured for a varying number of cores available to the web server. The results are shown in Figure 6. μC++ scales only up to 4 cores and given its low throughput result, is excluded from further experiments. Fred performs as good as ULib in this stress test, while Pthread shows a noticeable performance gap. Go also scales, but its throughput is even lower than Pthread. We have experimented with small modifications to Go's runtime that mimic the operation of Fred's I/O subsystem. It is possible to increase Go's throughput for this benchmark by up to 50%, closer to Pthread, but still clearly below ULib and Fred. However, it would be presumptuous to suggest such changes to production-level Go without wider and rigorous testing.

It is important to point out that scaling to such high core counts in this experiment is only possible when working around inherent scalability limitations in Linux's `epoll` event notification facility. ULib's cloned process architecture makes it immune to these limitations, while Pthread does not need separate event notifications at all. The fred runtime works around these limitations by using up to 8 poller freds, as explained in Section 4.2.2. A supplementary experiment is described in Section 5.2.4 to examine the details. Most importantly, these limitations are a characteristic of this extreme I/O stress test and do not apply to the benchmarks reported below or the application experiments reported in Section 6.

5.2.2 Connection Scalability. In a second experiment, the number of cores is fixed at 16 and the number of connections is scaled. The methodology for this experiment is slightly more involved. Given that TCP ports are limited to 16 bits, a single client operating from a single IP address can only establish around 60,000 connections. In order to run these experiments with a larger number of connections, aliases of the local `lo` network interface are created with private addresses. Using those aliases, multiple clients can communicate with a single local server. The results are shown in Figure 7. The Pthread implementation of the web server can only handle up to 30,000 connections

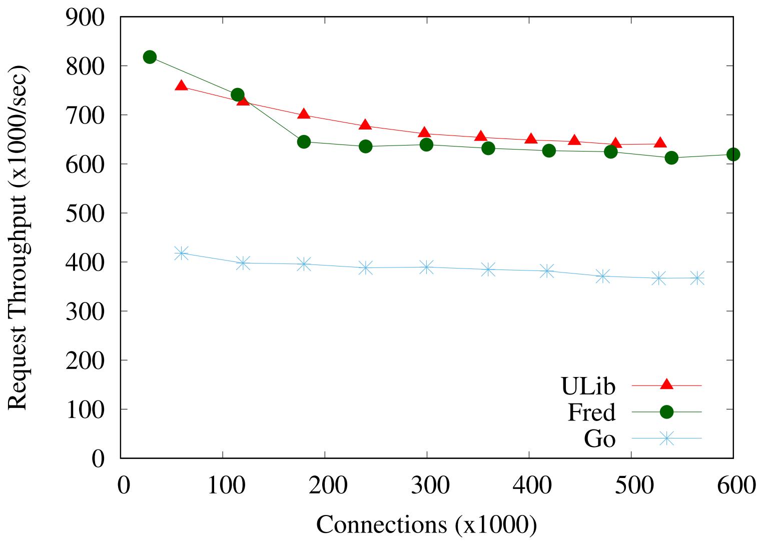


Fig. 7. Web Server (16 cores)

(threads) and it is thus not shown. Fred delivers similar performance to ULib, even with a single poller fred, while Go lags behind. The per-connection memory consumption for the threaded web server implementations were measured separately and range from 8 KB for Fred to 12 KB for Pthread.

5.2.3 Connection Churn. The last experiment in this section investigates the performance with high connection churn that might occur in a busy Internet server serving many small requests from many different clients. The 'weighttp' client [63] is used for this experiment, configured to create a new connection for each HTTP request. The results show attainable throughput for varying core counts in Figure 8 and demonstrate that ULib and Fred deliver comparable performance. Fred is the only thread-per-session web server that can sustain a throughput rate comparable to ULib and this is only possible, if it is partitioned into at least 4 event scopes for this experiment, as described in Section 4.2.3. It also clones the server socket across event scopes using the `SO_REUSEPORT` socket option. A supplementary experiment is described in the next section to examine the details. ULib uses the same underlying system-level mechanisms for scalability and in fact, partitions the system into N process copies with a separate FD table for each. Pthread and Go deliver unspectacular performance and the Go-based web server crashes beyond 16 cores.

5.2.4 I/O Bottlenecks. Two specific bottlenecks in Linux I/O processing have a significant impact on I/O performance.

First, the `epoll` event notification facility does not scale beyond 16 cores. At 32 cores across NUMA boundaries, it is necessary to partition FDs into multiple interest sets. With the Fred runtime, this is possible by using multiple poller freds. To demonstrate this effect, the experiment from Section 5.2.1 is repeated for 32 cores with a varying number of poller freds. The results in Figure 9a show that a minimum number of poller freds is necessary (4 in this case) to achieve good throughput, while additional poller freds do not affect throughput any further. However, more poller freds might slightly increase latencies, as discussed in Section 4.2.2.

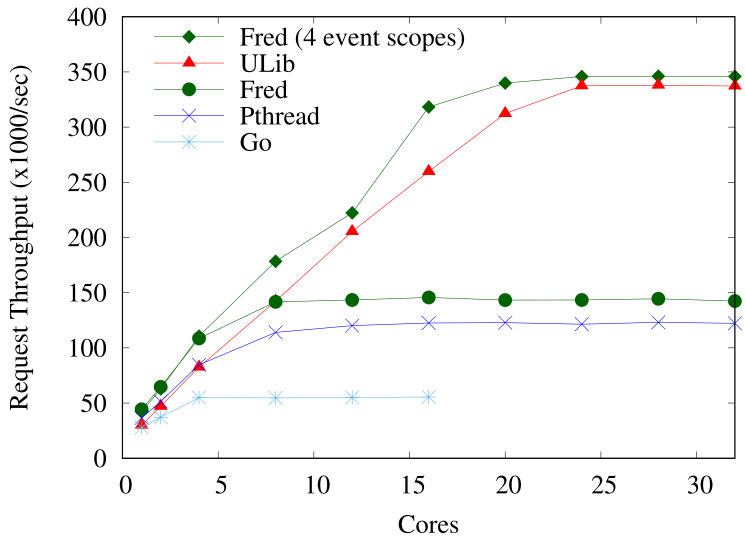
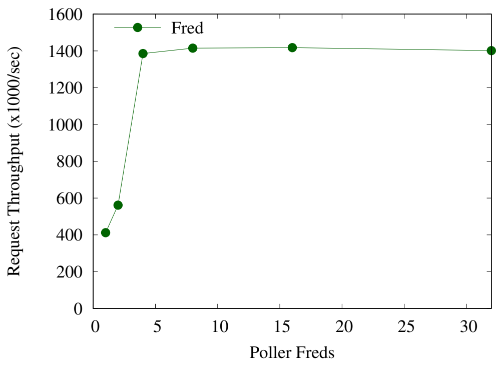
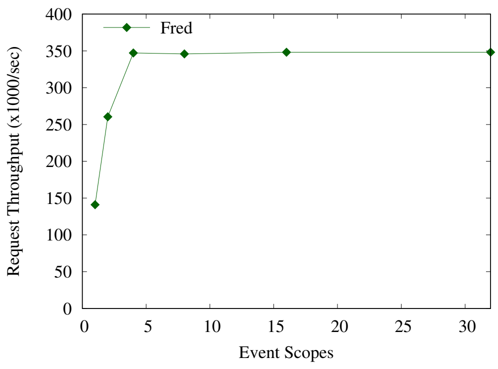


Fig. 8. Web Server (short-lived)



(a) Web Server (15K connections)



(b) Web Server (short-lived)

Fig. 9. Fred Configuration (32 cores)

The second bottleneck is the performance of the `accept` system calls, which is crucial when handling many short-lived connections. The only workaround for this bottleneck is splitting the set of FDs into multiple kernel tables, as facilitated by ULib or Fred's event scope feature. The experiment from Section 5.2.3 is repeated for 32 cores with a varying number of event scopes. The results in Figure 9b show that a minimum number of event scopes is necessary (4 in this case) to achieve good throughput, while additional event scopes do not affect throughput any further. However, multiple event scopes inhibit automatic load balancing, because FDs are not valid across event scopes (cf. Section 4.2.3).

Both of these bottlenecks are fundamental to Linux and there is no silver bullet solution. The Fred runtime provides workarounds, but then again, these workarounds are probably not needed in real applications. Most likely, these bottlenecks are only relevant in stress-testing benchmarks.

The application experiments reported in the next section do not use either workaround. Instead, they are run with a single poller fred and a single event scope.

5.2.5 Summary. The overarching conclusion drawn from these experiments is that user threading can provide competitive I/O performance, while system threads show clear limitations. Of all user-level threading systems, only the fred runtime is efficient and scalable enough to perform on par with event-driven execution for these stress-testing I/O benchmarks.

6 APPLICATION PERFORMANCE

Memcached is an in-memory key-value store, primarily used as an object cache for the results of dynamic queries in web applications. It is a mature software system that is deployed in a broad range of large-scale production environments [42]. The server is designed as an event-driven state machine, which is replicated across multiple system threads to utilize processor cores. It thus supports multi-core execution and shared mutable data, but has a simpler software structure than contemporary web servers or database servers. These characteristics make it the ideal candidate for studying different runtime paradigms. All experiments use the 'mutilate' load generator [38] with its synthetic recreation of a Facebook workload described as "ETC" in the literature [8].

We have converted the original Memcached implementation with a minimal set of modifications to the thread-per-session execution model without loosing essential functionality. In particular, the core state machine for processing requests is still present in the modified version, but it is synchronously driven by a separate thread (or fred) for each connection. The thread-per-session version of Memcached is used to evaluate the performance of system threads ('Pthread') and user-level freds in comparison with the original event-driven implementation ('Vanilla') and Arachne's version. Arachne also uses Memcached for comparison [50], but Arachne lacks user-level I/O blocking support and correspondingly, its Memcached version uses a thread-per-request model. Furthermore, the Arachne version of Memcached does not actually use Arachne's user-level synchronization operations.

6.1 Memcached Transformation

For reproducibility, the transformation of Memcached to thread-per-connection processing is documented in 10 steps, each of which results in a working version with good performance. The combined changes of these modifications amount to adding ~1000 lines of code, while removing ~400 lines and the dependency on libevent [2]. This results in a version that uses one system thread per connection, which is used to run the Pthread experiments reported below. One more update replaces system threads with freds by replacing another ~500 lines. In total, this changes about 5% of Memcached's direct code base. These changes are applied to Memcached 1.5.7, which is the version used for the experimental evaluation. The modifications remove some functionality and protocol operations that are not performance-critical, but would increase the complexity of porting: per-thread statistics aggregation and the monitoring commands 'lru_crawler metadump' and 'watch'. Also, Memcached uses stop-the-world synchronization to halt all operations during the resizing of its central hash table. The fred runtime provides similar functionality, but might incur a somewhat higher latency (not investigated yet). While this scenario does not matter in reality, since hash table resizing is extremely rare and never undone, it might affect short-term benchmark experiments. Therefore, this part is avoided by always pre-loading and warming the server.

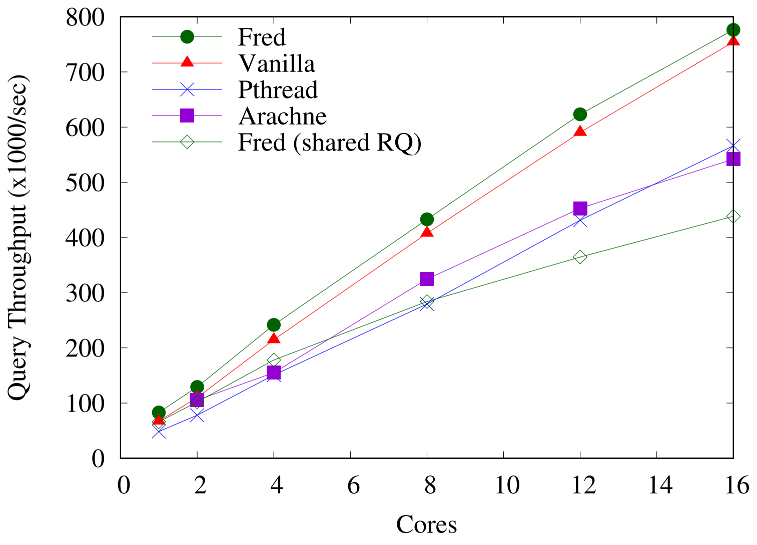


Fig. 10. Memcached (1,200 connections)

6.2 Throughput

The next set of experiments is also run on the 4-socket AMD machine and investigates basic efficiency and throughput performance.

6.2.1 Cores / Connections. The server uses up to 16 cores, while the remaining 48 cores are used for unrestricted closed-loop load generation without pipelining and with an update ratio of 0.1. This provides ample work for the server and all Memcached variants report full CPU utilization during the experiments. The first experiment is run with 1,200 connections, similar to Arachne's settings.[2] The per-connection memory consumption is approximately 12 KB for Vanilla and Arachne, while the thread-per-session versions use about 16 KB memory per connection. Figure 10 shows that only Fred has the same execution efficiency as Vanilla and thus comparable performance. A second experiment increases the connection count to 24,000 and clearly documents the limitations of Arachne's approach in terms of I/O. As shown in Figure 11, at a higher number of connections, Arachne is unable to deliver competitive performance. As part of these experiments, we also compare to a version of the Fred runtime that is limited to using a single shared ready-queue for scheduling. The reduced efficiency and performance indirectly demonstrate the benefits of multi-queue scheduling in Fred.

6.2.2 Update Ratio. The previous experiments use an update ratio of 0.1, which is a reasonable test value for Memcached, because it is intended as an object cache. However, as a matter of principle, it is also interesting to see how performance changes in relation to this parameter, because it affects the locking intensity in the server. Furthermore, Memcached with an increased update ratio can provide a preliminary glimpse of the performance behavior of a database system with a transactional workload. Thus, in the next experiment, the number of cores is fixed at 16 and the number of connections at 1,200, while the update ratio is varied. The results are shown in Figure 12 and are generally consistent with the results shown in Figure 10. However, at increased update

[2]Table 3 in the Arachne paper [50] seems to describe a workload of more than 25,000 connections, but those experiments have actually only used 1280 connections in total. This is a confirmed typo in the Arachne paper.

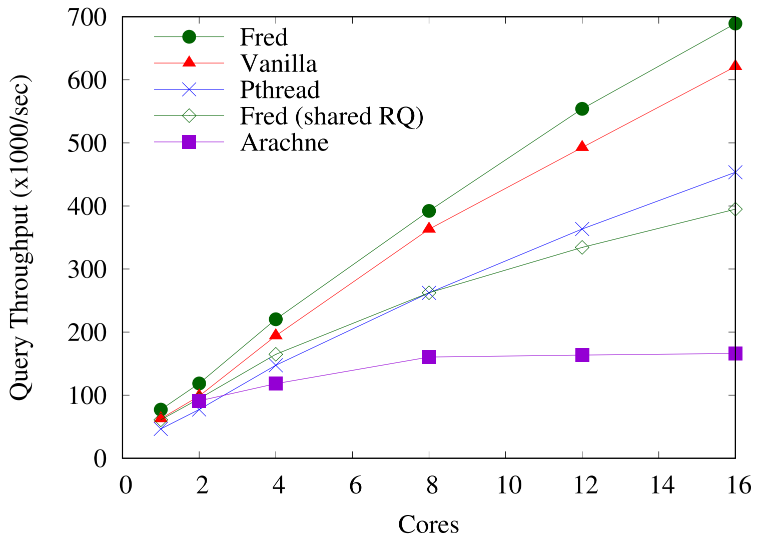


Fig. 11. Memcached (24,000 connections)

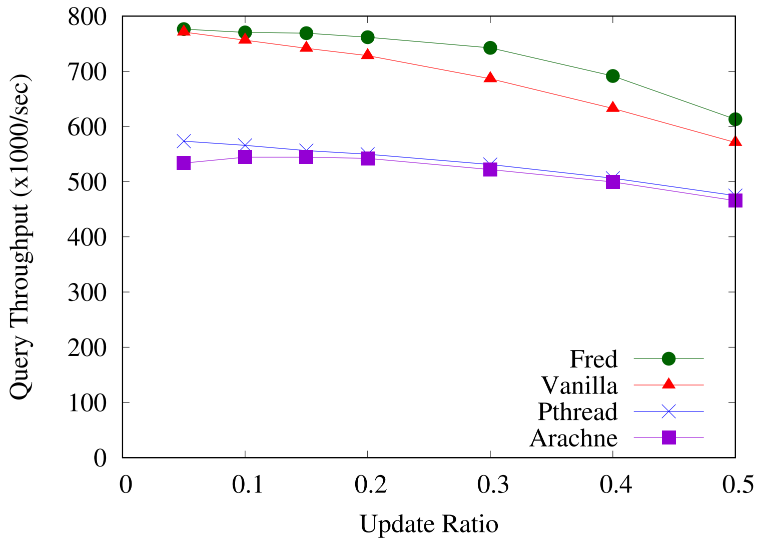


Fig. 12. Memcached / Updates (1,200 connections)

ratios, Fred appears to have a slight edge over Vanilla. We conjecture that user-level locking is cheaper than system-level locking, but future work is needed to determine the significance of this effect.

6.2.3 Skewed Workload. The next experiment is designed to test load-balancing. One client uses 16 connections to produce a background load of approximately 500K queries/sec that is evenly distributed across 16 server cores, because of the default round-robin assignment of connections (or

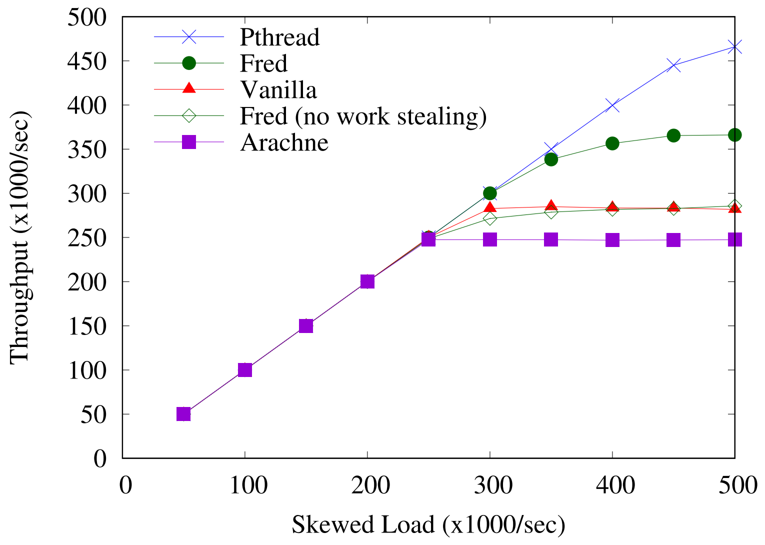


Fig. 13. Memcached / Skew (1,200 connections)

freds) to cores. A second client creates another workload that increases from 50K to 500K queries/sec. And evenly balanced system can handle roughly 1000K queries/sec under these circumstances (tested separately). However, the second clients uses only 8 connections, such that 8 cores receive an increasingly higher request rate. It is expected that the first 250K queries/sec of extra workload can be handled by these 8 cores. However, additional throughput can only be achieved via load balancing. The experiment measures the portion of the extra workload that can be accommodated. The results are shown in Figure 13.

Vanilla Memcached does not implement any load balancing. It can still handle a small part of the extra load, because the system limitation of 1000K is somewhat approximate. A test version of Fred only uses processor-local ready-queues without work stealing. It shows the same performance as Vanilla. The actual Fred runtime with work stealing can accommodate a substantial portion of the extra load. This demonstrates the effectiveness of load balancing over simply using local ready-queues. Pthread shows excellent performance, because for this low number of system threads, the kernel scheduler can provide superior work stealing and load balancing. Arachne is unable to handle any of the extra load in this experiment. We suspect that this experiment hits a corner case, but a detailed investigation is outside of the scope of this paper.

6.3 Tail Latency

The Arachne paper [50] presents an interesting observation: Packet processing in the kernel network stack can cause high tail latency for Memcached and presumably other applications, which can be mitigated by user-level load balancing. This section reports the attempt to reproduce Arachne's latency experiment [50]. Client and server machines are dual-socket Intel E5-2620 with 64 GB RAM, 12 cores, 2xHT, but only one of each pair of hyperthreads is used. They run Ubuntu Linux 16.04 with kernel version 4.15.0-50-generic and GCC 5.4.0. Networking is done using Mellanox 10 Gbit/s NICs, configured with 16 RX rings and receive-side scaling (RSS), and a corresponding Mellanox switch. 8 client machines create an orchestrated workload, comprised of 1536 connections in total, at specified aggregate rates with an update ratio of 0.03. One dedicated client creates a

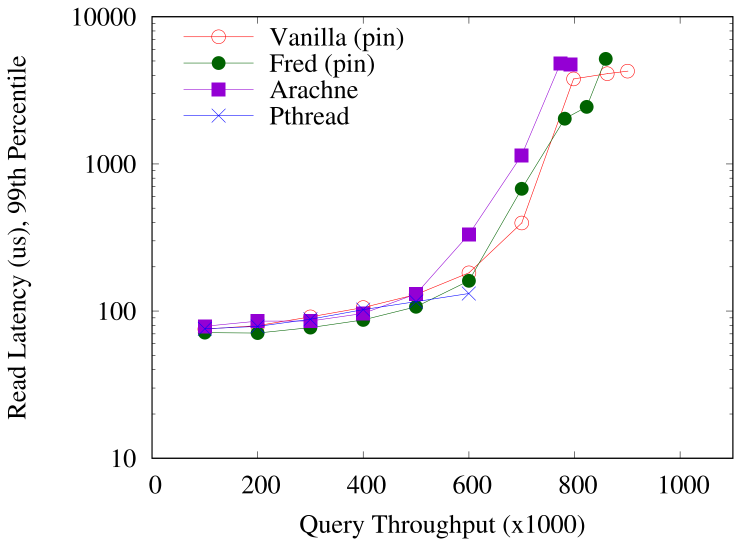


Fig. 14. Memcached / Tail Latency (1,536 connections)

low-rate (1000 queries/sec) request stream that is used to measure end-to-end service latency. The settings closely resemble the setup from Arachne's evaluation.[3] The results show the 99th percentile read latency (in logscale) on the Y-axis for the actually achieved throughput on the X-axis. Each data point shows the average of 20 runs and the results are shown in Figure 14. However, our results do not confirm the earlier findings. In our setup, which uses a more recent Linux kernel version and drivers, all variants show similar tail latency behavior. Interestingly, even Pthread is competitive, but only up to a particular throughput. When repeating the same experiment with a 5x higher connection count, only Fred has similar performance to Vanilla, as shown in Figure 15. Both Arachne and Pthread produce higher tail latencies at lower throughput and have a lower maximum throughput. The discrepancy between our and Arachne's results is studied further in Appendix B.

7 SUMMARY AND CONCLUSION

The results of our evaluation are summarized in Table 1. Only the fred runtime delivers the complete set of desirable characteristics. The columns termed *Scalability* and *Efficiency* directly refer to the experiment results reported in Section 5.1. The findings reported in column *Idle-Sleep* have been obtained by code inspection and are consistent with the observed behavior during the efficiency experiments. *Scalable I/O Blocking* is investigated in Section 5.2. The experiments in Section 6.2.1 and 6.2.2 confirm that the previous observations apply in a real application. *Load Balancing* is investigated in Section 6.2.3. Tail latency in a real-world hardware setup (cf. Section 6.3) completes the overall performance picture.

The overarching conclusion from our work is that user-level threading, when done right, can deliver competitive performance to event-driven execution, while also offering the benefits of load balancing. We present the first user-level M:N threading runtime that delivers this level of efficiency and scalability – better than existing runtimes and comparable with a best-in-class event-based system. These findings are based on a thorough evaluation of several user threading runtimes in comparison with system threads and event-driven execution using both ULib and Memcached.

[3]See Footnote 2 on Page 19.

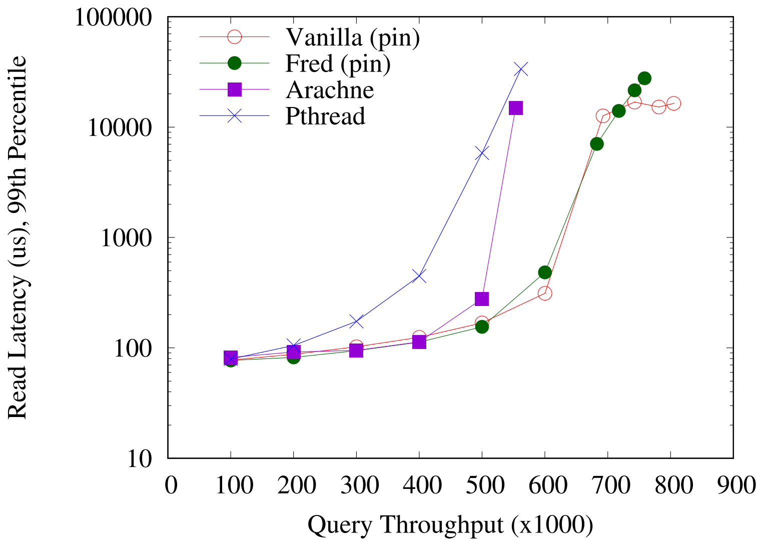


Fig. 15. Memcached / Tail Latency (7,680 connections)

Table 1. Summary of Evaluation

System Name	Scalability	Efficiency	Idle-Sleep	Scalable I/O	Load Balancing
Fred	yes	yes	yes	yes	yes
Event-Driven	yes	yes	yes	N/A	no
Pthread	yes	yes	yes	limited	yes
Go 12.2	yes	yes	1 core loop/sleep	good	yes
Arachne	yes	moderate	1 core spin	no	yes
μC++ 7.0.0	no	no	yes	poor	yes
Boost Fiber 1.69	yes	no	all cores spin	no	yes
Libfiber	yes	yes	all cores loop/sleep	no	not tested
Mordor	moderate	moderate	not tested	not tested	not tested
Qthreads 1.13	yes	no	all cores spin	no	yes

In the long run, this work might hint at a new division of work between kernel and user-level. The preemptive kernel scheduler can focus on resource management and fairness between applications, while variations of our user-level scheduler would be geared towards the specific needs of different types of applications. Such an approach could also include cross-layer optimizations across kernel- and user-level scheduling.

ACKNOWLEDGEMENTS

The work reported in this paper has been partially supported by Huawei, as well as the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Canada Foundation for Innovation (CFI).

REFERENCES

[1] Lock and semaphore icons designed by Freepik (https://www.flaticon.com/authors/freepik) from Flaticon (www.flaticon.com). Licensed via Flaticon Basic License.

[2] libevent. http://libevent.org/. [Online; accessed 2019-12-27].

[3] libev. http://libev.schmorp.de/. [Online; accessed 2019-12-27].

[4] libuv. http://libuv.org/. [Online; accessed 2019-12-27].

[5] Adya, A., Howell, J., Theimer, M., Bolosky, W. J., and Douceur, J. R. Cooperative Task Management Without Manual Stack Management. In *Proceedings of USENIX ATC* (2002), pp. 289–302.

[6] Akka Toolkit. https://akka.io/. [Online; accessed 2019-12-27].

[7] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism. In *Proceedings of ACM SOSP* (1991), pp. 95–109.

[8] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of SIGMETRICS* (2012), pp. 53–64.

[9] Barghi, S. uThreads: Concurrent User Threads in C++(and C). https://github.com/samanbarghi/uThreads. [Online; accessed 2019-12-27].

[10] Barroso, L., Marty, M., Patterson, D., and Ranganathan, P. Attack of the killer microseconds. *Commun. ACM 60*, 4 (Mar. 2017), 48–54.

[11] Blumofe, R. D., and Leiserson, C. E. Scheduling multithreaded computations by work stealing. *J. ACM 46*, 5 (Sept. 1999), 720–748.

[12] Boost.Fiber. https://www.boost.org/doc/libs/1_69_0/libs/fiber/doc/html/index.html. [Online; accessed 2019-12-27.

[13] Brown, T., Ellen, F., and Ruppert, E. A General Technique for Non-blocking Trees. In *Proceedings of ACM PPoPP '14* (2014), pp. 329–342.

[14] Buhr, P. A., Ditchfield, G., Stroobosscher, R. A., Younger, B. M., and Zarnke, C. R. μC++: Concurrency in the Object-Oriented Language C++. *Software: Practice and Experience 22*, 2, 137–172.

[15] Buntinas, D., Mercier, G., and Gropp, W. Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem. In *Proceedings of CCGRID* (2006), pp. 521–530.

[16] C++ Actor Framework. https://actor-framework.org/. [Online; accessed 2019-12-27].

[17] Casazza, S. ULib Application Development Framework. https://github.com/stefanocasazza/ULib. [Online; accessed 2019-12-27].

[18] Dice, D. Malthusian Locks. In *Proceedings of EuroSys* (2017), pp. 314–327.

[19] Dmitry Vyukov. Intrusive MPSC node-based queue. http://www.1024cores.net/home/lock-free-algorithms/queues/intrusive-mpsc-node-based-queue. [Online; accessed 2019-12-27].

[20] Elmeleegy, K., Chanda, A., Cox, A. L., and Zwaenepoel, W. Lazy Asynchronous I/O for Event-driven Servers. In *Proceedings of USENIX ATC* (2004), pp. 21–21.

[21] Erb, B. Concurrent Programming for Scalable Web Architectures. Diploma thesis, Institute of Distributed Systems, Ulm University, April 2012.

[22] Erlang Programming Language. https://www.erlang.org/. [Online; accessed 2019-12-27].

[23] folly::fibers. https://github.com/facebook/folly/. [Online; accessed 2019-12-27].

[24] Gasiunas, V., Dominguez-Sal, D., Acker, R., Avitzur, A., Bronshtein, I., Chen, R., Ginot, E., Martinez-Bazan, N., Müller, M., Nozdrin, A., Ou, W., Pachter, N., Sivov, D., and Levy, E. Fiber-based Architecture for NFV Cloud Databases. *Proc. VLDB Endow. 10*, 12 (Aug. 2017), 1682–1693.

[25] Glozer, W. wrk - a HTTP benchmarking tool. https://github.com/wg/wrk. [Online; accessed 2019-12-27].

[26] The Go Programming Language. https://www.golang.org/. [Online; accessed 2019-12-27].

[27] Graham, R. The C10M Problem. http://c10m.robertgraham.com/p/manifesto.html, 2013. [Online; accessed 2019-12-27].

[28] Gramoli, V. More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. In *Proceedings of ACM PPoPP 2015* (2015), pp. 1–10.

[29] Guiroux, H., Lachaize, R., and Quéma, V. Multicore Locks: The Case is Not Closed Yet. In *Proceedings of USENIX ATC* (2016), pp. 649–662.

[30] Haller, P., and Odersky, M. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science 410*, 2 (2009), 202 – 220. Distributed Computing Techniques.

[31] Harji, A. S., Buhr, P. A., and Brecht, T. Comparing High-performance Multi-core Web-server Architectures. In *Proceedings of SYSTOR* (2012), pp. 1:1–1:12.

[32] Callback Hell. https://en.wiktionary.org/wiki/callback_hell. [Online; accessed 2019-12-27].

[33] Hewitt, C., Bishop, P., and Steiger, R. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of IJCAI* (1973), pp. 235–245.

[34] Hoare, C. A. R. Communicating Sequential Processes. *Communications of the ACM 21*, 8 (Aug. 1978), 666–677.

[35] Ian Lance Taylor. Split Stacks in GCC. https://gcc.gnu.org/wiki/SplitStacks. [Online; accessed 2019-12-27].

[36] IEEE and The Open Group. Standard for Information Technology–Portable Operating System Interface Base Specifications, Issue 7, Sep. 2016. https://doi.org/10.1109/IEEESTD.2016.7582338 - see Section 2.14 File Descriptor Allocation.
[37] Kegel, D. The C10K Problem. http://www.kegel.com/c10k.html, 1999. [Online; accessed 2019-12-27].
[38] Leverich, J. Mutilate. https://github.com/leverich/mutilate. [Online; accessed 2019-12-27].
[39] Li, S., Lim, H., Lee, V. W., Ahn, J. H., Kalia, A., Kaminsky, M., Andersen, D. G., O, S., Lee, S., and Dubey, P. Full-Stack Architecting to Achieve a Billion-Requests-Per-Second Throughput on a Single Key-Value Store Server Platform. *ACM Trans. Comput. Syst. 34*, 2 (Apr. 2016), 5:1–5:30.
[40] A User Space Threading Library Supporting Multi-Core Systems. https://github.com/brianwatling/libfiber/. [Online; accessed 2019-12-27].
[41] Lozi, J.-P., David, F., Thomas, G., Lawall, J., and Muller, G. Remote Core Locking: Migrating Critical-section Execution to Improve the Performance of Multithreaded Applications. In *Proceedings of USENIX ATC* (2012), pp. 6–6.
[42] Memcached. http://www.memcached.org/. [Online; accessed 2019-12-27].
[43] Mordor I/O Library. https://github.com/mozy/mordor/. [Online; accessed 2019-12-27].
[44] Nakashima, J., and Taura, K. MassiveThreads: A Thread Library for High Productivity Languages. In *Concurrent Objects and Beyond*, G. Agha, A. Igarashi, N. Kobayashi, H. Masuhara, S. Matsuoka, E. Shibayama, and K. Taura, Eds., vol. 8665 of *Lecture Notes in Computer Science*. Springer, pp. 222–238.
[45] Osterhout, J. Why Threads Are A Bad Idea (for most purposes). Invited talk at USENIX ATC, 1996. https://web.stanford.edu/~ouster/cgi-bin/papers/threads.pdf. [Online; accessed 2019-12-27].
[46] Pariag, D., Brecht, T., Harji, A., Buhr, P., and Shukla, A. Comparing the Performance of Web Server Architectures. In *Proceedings of EuroSys* (2007), pp. 231–243.
[47] Pony Programming Language. https://www.ponylang.org/. [Online; accessed 2019-12-27].
[48] Proto Actor. http://proto.actor/. [Online; accessed 2019-12-27].
[49] GNU Pth. https://www.gnu.org/software/pth/. [Online; accessed 2019-12-27].
[50] Qin, H., Li, Q., Speiser, J., Kraft, P., and Ousterhout, J. Arachne: Core-aware Thread Management. In *Proceedings of OSDI* (2018), pp. 145–160.
[51] Quasar. http://www.paralleluniverse.co/quasar/. [Online; accessed 2019-12-27].
[52] Scaling in the Linux Networking Stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt. [Online; accessed 2019-12-27].
[53] Roghanchi, S., Eriksson, J., and Basu, N. Ffwd: Delegation is (Much) Faster Than You Think. In *Proceedings of SOSP* (2017), pp. 342–358.
[54] Schillings, B. Be Engineering Insights: Benaphores. https://www.haiku-os.org/legacy-docs/benewsletter/Issue1-26.html. [Online; accessed 2019-12-27].
[55] Segmented Stacks in LLVM. https://llvm.org/docs/SegmentedStacks.html. [Online; accessed 2019-12-27].
[56] State Threads Library. http://state-threads.sourceforge.net/. [Online; accessed 2019-12-27].
[57] TechEmpower, Inc. Web Framework Benchmarks. https://www.techempower.com/benchmarks/#section=data-r17&hw=ph&test=plaintext. [Online; accessed 2019-12-27].
[58] The Linux man-pages project. clone, __clone2 - create a child process. http://man7.org/linux/man-pages/man2/clone.2.html. See CLONE_FILES flag. [Online; accessed 2019-12-27].
[59] Turner, P. User-level threads....... with threads. Talk at Linux Plumbers Conference, 2013. http://www.linuxplumbersconf.org/2013/ocw/proposals/1653. [Online; accessed 2019-12-27].
[60] Valialkin, A. Fast HTTP Package for Go. https://github.com/valyala/fasthttp. [Online; accessed 2019-12-27].
[61] von Behren, R., Condit, J., and Brewer, E. Why Events Are a Bad Idea (for High-concurrency Servers). In *Proceedings of HOTOS* (2003), pp. 4–4.
[62] von Behren, R., Condit, J., Zhou, F., Necula, G. C., and Brewer, E. Capriccio: Scalable Threads for Internet Services. In *Proceedings of ACM SOSP* (2003), pp. 268–281.
[63] weighttp. https://github.com/lighttpd/weighttp. [Online; accessed 2019-12-27].
[64] Welsh, M., Culler, D., and Brewer, E. SEDA: An Architecture for Well-conditioned, Scalable Internet Services. In *Proceedings of ACM SOSP* (2001), ACM, pp. 230–243.
[65] Wheeler, K. B., Murphy, R. C., and Thain, D. Qthreads: An API for Programming with Millions of Lightweight Threads. In *Proceedings of IEEE IPDPS* (April 2008), pp. 1–8.
[66] Windows Fibers. https://docs.microsoft.com/en-us/windows/desktop/ProcThread/fibers. [Online; accessed 2019-12-27].

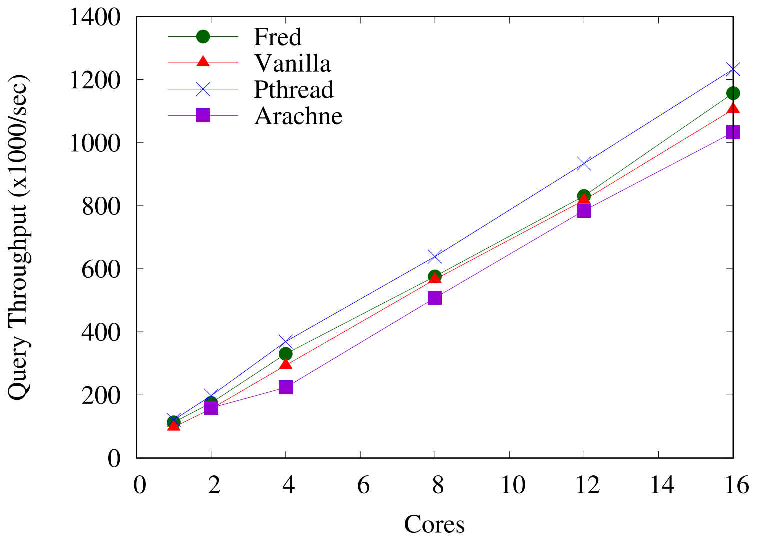


Fig. 16. Memcached (1200 connections, pipelining)

A MEMCACHED

This experiment is a variation of the experiments reported in Section 6.2.1. Increasing client-side pipelining to 16 results in more continuous traffic for each connection and thus reduces the overall load pressure. In this case, shown in Figure 16, all variants perform similarly.

B ARACHNE

The experiment described in Section 6.3 does not reproduce Arachne's striking results showing reduced tail latency with user-level threading [50]. We thus report a more detailed investigation of these results to illustrate how difficult it is to obtain an accurate understanding of full-system performance and in turn, how easy it is to arrive at premature conclusions. One issue missing from Arachne's evaluation methodology is any attempt to understand the reasons for vanilla Memcached's higher tail latency. The higher latency distribution measured for the original Memcached implementation is not accompanied by lower throughput. Therefore, those latencies are not caused by processing inefficiencies, but must be caused by queueing effects. Consequently, we assume that parallel packet processing in the network stack leads to logical reordering and latency distortions.

B.1 Packet Processing

Specifically, we conjecture that these latencies are caused, at least partially, by head-of-line blocking along the parallel packet processing path in the Linux kernel. If parallel packet processing is responsible for some of the latencies, Linux's receive-flow steering (RFS) [52] should reduce packet dispersion and thus reordering. However, for a multi-threaded application, RFS is more effective with system thread pinning to avoid moving targets for RFS. In this context, thread pinning establishes an affinity of each system thread to a specific processor core. Therefore, we compare original Memcached with and without RFS and thread pinning. A key difference is that Arachne's results were obtained using Ubuntu Linux 14.04. Therefore, we repeat the first experiment from Section 6.3 for vanilla Memcached, but using Ubuntu Linux 14.04 with kernel version 4.4.0-134-generic and

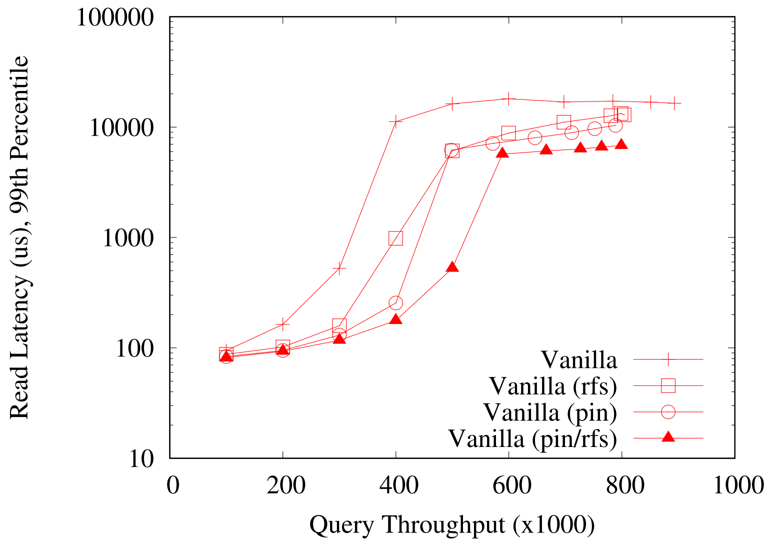


Fig. 17. Memcached - Packet Processing, 14.04, RX8

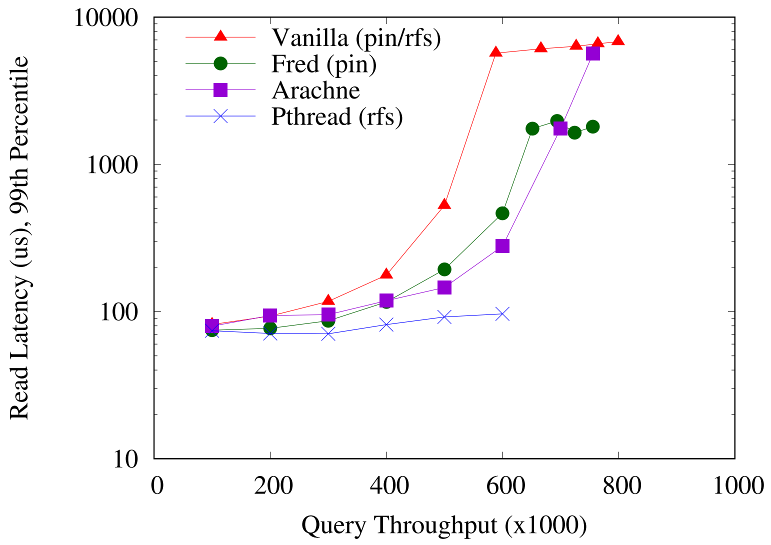


Fig. 18. Memcached - Comparison, 14.04, RX8

GCC 4.8.4. In this version, the correspondingly older Mellanox driver only supports 8 RX rings for receive-side scaling (RSS).

The results are shown in Figure 17. Both RFS and thread pinning separately and in combination improve Memcached's latency profile at the expense of somewhat reduced maximum throughput. This corroborates the conjecture that queueing effects from parallel packet processing contributes to higher tail latency.

The next experiment compares the latency and throughput behavior of different Memcached variants. We present the respective best result for each variant in Figure 18. Fred benefits from thread pinning in this scenario, but not RFS, while Pthread performs best with RFS enabled. Arachne uses thread pinning inherently through its core arbiter and does not benefit from RFS. While the vanilla version of Memcached with RFS and thread pinning performs better than without, the additional load balancing capabilities of multi-threading can still improve tail latency. Using conventional system threads provides striking performance in this experiment – up to a certain maximum throughput. Arachne operates as documented in the literature and shows a better latency profile than Vanilla. However, Fred performs comparably to Arachne, but without any of Arachne's inherent limitations.

Another observation is concerned with the fact that Arachne has not been compared to a system thread variant of Memcached. If a tight latency distribution is much more important than attainable throughput or connection scalability, Figure 18 shows that directly using system threads is an attractive alternative. The explanation for this omission is rooted in Arachne's design limitations. Because Arachne does not support user-level I/O blocking, it can only be used for thread-per-request execution. Consequently, its version of Memcached does not support the thread-per-session model that is necessary for direct execution with system threads. An important rationale for thread-per-session programming is automatic state capture across multiple requests in a session, which is not supported by Arachne's approach to user-level threading.

B.2 Receive Queues

The Mellanox driver in Ubuntu 14.04 only supports 8 RX queues, while the experiments use 12 processor cores. Therefore, multiple cores are receiving packets from the same RX queue, which gives rise to the concern about head-of-line blocking. The corresponding driver in Ubuntu 16.04 supports up to 24 RX queues, which allows testing this hypothesis. When rerunning the Memcached experiments with 8 RX queues on Ubuntu 16.04, the results shown in Figures 19 and 20 show slightly higher performance overall, but are generally similar to the previous ones shown in Figures 17 and 18. Note that the Ubuntu 16.04 kernel (version 4.15.0-50-generic) runs with most Spectre/Meltdown protections disabled to make the results comparable.

However, when changing the driver configuration to 16 RX queues, exceeding the number of used processor cores, vanilla Memcached's tail latency is dramatically reduced, at least when also using thread pinning. At the same time, RFS becomes largely irrelevant. The results for vanilla Memcached are shown in Figure 21. A comparison between variants is already presented as Figure 14 in Section 6.3. As pointed out there, the overall latency profiles of the different Memcached variants with 16 RX queues are fairly similar and they only differ in maximum attainable throughput, which is consistent with the results reported in Section 6.2.

These results do not invalidate the fundamental observation that improved load balancing through user-level threading can reduce tail latency in certain situations. However, it puts that observation in perspective by demonstrating that high tail latency may be less inevitable than suggested previously.

As pointed out in Section 2, attention to detail matters. The localhost experiments reported in Sections 5.2 and 6.2 avoid certain effects that only arise when including real networking hardware and device drivers. On the other hand, the full stack including hardware introduces other side effects that are not always obvious. The corresponding system-level configuration parameters then also need to be taken into account when assessing system performance.

Received October 2019; revised December 2019; accepted January 2020

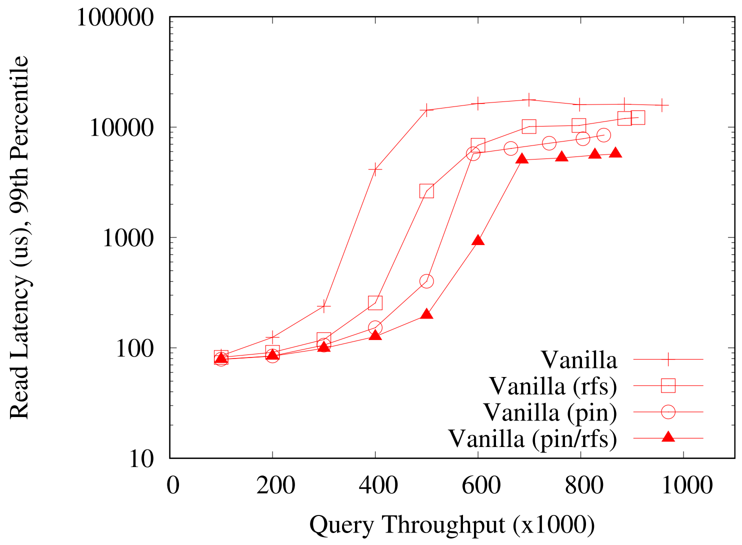


Fig. 19. Memcached - Packet Processing, 16.04, RX8

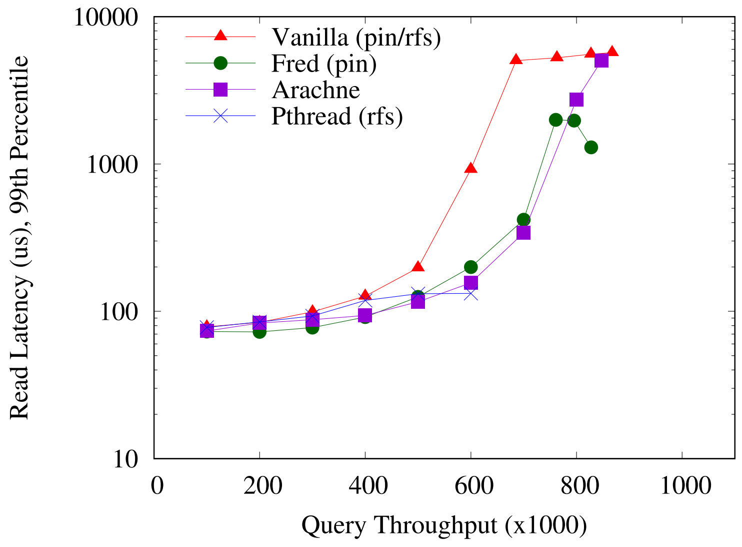


Fig. 20. Memcached - Comparison, 16.04, RX8

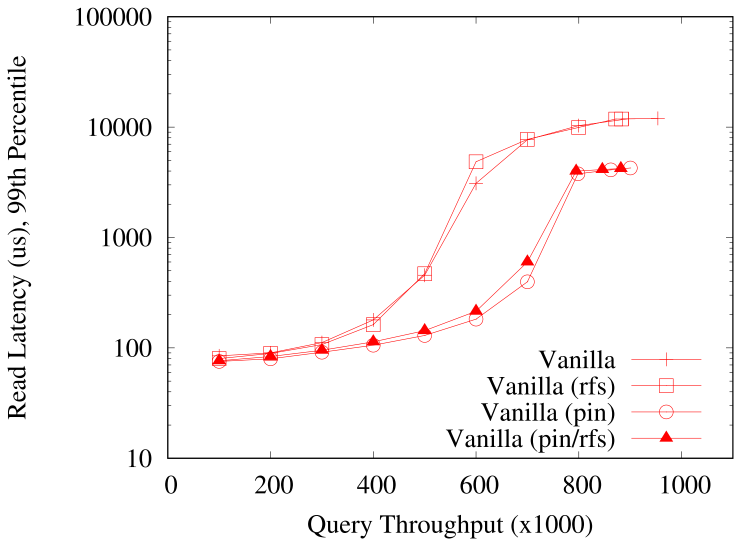


Fig. 21. Memcached - Packet Processing, 16.04, RX16