Florian Schmaus, Florian Fischer, Timo Hönig, Wolfgang Schröder-Preikschat

Modern Concurrency Platforms Require Modern System-Call Techniques

Technical Report CS-2021-02

November 2021
Modern Concurrency Platforms Require Modern System-Call Techniques

Florian Schmaus, Florian Fischer
Wolfgang Schröder-Preikschat
Friedrich-Alexander-University
Erlangen-Nürnberg (FAU)
Erlangen, Germany
{flow,f.fischer,wosch}@cs.fau.de

Timo Hönig
Ruhr University Bochum (RUB)
Bochum, Germany
timo.hoenig@rub.de

Abstract—Writing well-maintainable parallel programs that efficiently utilize many processor cores is still a significant challenge. Threads are hard to use, and so are event-based schemes. Furthermore, threads are affected by the blocking anomaly, that is, the loss of parallelism when threads execute a blocking system call—often resulting in low core utilization and unnecessarily high response times. This paper introduces pseudo-blocking system calls built upon modern asynchronous queue-based system-call techniques (like Linux’s io_uring) circumventing the blocking anomaly. They are similar to Go's programming model, where one develops against a blocking interface to keep the code structure clean. However, instead of using synchronous non-blocking system calls as the underlying technique, our approach internally uses an asynchronous queue-based interface. We further present a novel architecture for concurrency platforms, like Cilk and Go, enabling low latencies and high throughput via pseudo-blocking system calls. Finally, we discuss future OS enhancements that would improve our proposed architecture. We implemented and evaluated a concurrency platform based on the concept of pseudo-blocking system calls. Our platform can outperform state-of-the-art systems like Go by \(1.17 \times\) in a file-content search benchmark. It is able to increase the throughput of a echo-server benchmark by 4% when compared to Go, and by 17.8% when compared to Rust’s Tokio while improving the tail latency.

I. INTRODUCTION

Current and future computer architectures are equipped with more and more processor cores. This imposes new challenges to application developers and system software designers to keep the plentiful hardware resources reasonably utilized. At the same time, applications are required to perform a large amount of concurrent input and output operations, not only to persistent storage but also with a considerable amount of remote communication partners. This leads to the question of how software that can efficiently use the available hardware resources while allowing for concurrent I/O operations has to be designed and developed. Furthermore, to keep maintainability, all of that should be feasible without increasing the perceived complexity of the software. This is not only true for the application layer but, by transitivity, also applies to system software.

Listing 1: Implementation of a sequential network server

```c
void server(int listen_socket) {
  int fd = accept(listen_socket);
  read(fd, input);
  auto output = process(input);
  write(fd, output);
  close(fd);
}
```

Listing 1 shows a simple implementation of a network server. Since the code uses blocking system calls for I/O operations, it is not possible to serve multiple clients with a single-core system. On the other hand, the control flow follows a clear path from the beginning to the end of the function. This makes the implementation simple and easy to grasp.

If we want to serve multiple clients concurrently, then we could implement the server in an event-driven fashion. Listing 2 shows such an implementation based on the reactor [1] design pattern. While this implementation can serve multiple clients concurrently using a single processor core, it is considerably harder to understand and maintain. This is because the application logic is now scattered around multiple places [2], and the control flow is no longer explicit but implicit and potentially depends
eventLoop loop;

void on_accept(fd) { loop.register(fd, READ); }

void on_read(fd, input) {
    output = process(input);
    loop.register(fd, WRITE, output);
}

void on_write(fd, output) { close(fd); }

void server(int listen_socket) {
    loop.register(listen_socket, ACCEPT);
    loop.run();
}

Listing 2: Implementation of an event-driven network server

on shared state. Furthermore, parallelizing the server, i.e. ensuring that it can run on multiple cores in parallel, requires additional effort. Since an event loop is strictly single-threaded in the reactor pattern, parallelizing a reactor-based program means either creating multiple event loops, ideally one for every core. The more radical alternative entails modifying the code by changing the employed design pattern from reactor to a proactor [3], probably in conjunction with a thread pool. However, threads are hard to use [4].

Eventually, this does not solve all underlying causes that could lead to an underutilization of the available hardware resources. Consider what happens if process() in line 4 of Listing 2 blocks, e.g. because the (non-visible) implementation performs a blocking system call which is waiting for disk I/O to be completed. In this case, a worker thread from the pool of threads would be temporarily absent. If there is further outstanding work in the pool’s queue, while all other threads of the pool, but the blocked one, are busy, no entity will handle the outstanding work, even though there may be computational cores idle and hence available. This is referred to as the blocking anomaly [5]. Countermeasures against the blocking anomaly, like scheduler activations [6], often require complex interaction between the operating system and the application and impose a high overhead, limiting scalability.

We want to supply developers with a way to write parallel and scalable applications based on accessible and comprehensible source code. The dilemma appears to be that scalability in this context implies non-blocking I/O operations and the usage of those results in complex code.

One key enabler towards scalability are concurrency platforms like Go, Cilk [7, 8], and OpenMP. They allow to write structured parallel code that is able to make use of the plentiful available computational resources at runtime. However, they often lack scalable abstractions for efficient I/O operations, which, for example, results in over-provisioning the system with kernel-level threads.

In order to solve the previously mentioned dilemma, this paper presents the following contributions: 1) we survey existing system-call techniques, 2) we propose pseudo-blocking system calls based on modern asynchronous queue-based system-call techniques, 3) we present a distributed and scalable architecture for concurrency platforms to support pseudo-blocking system calls, 4) we implemented and evaluated this architecture in our research concurrency platform (EMPER), and 5) we discuss promising future research directions, including a novel I/O stealing technique.

The remainder of this paper is structured as follows: We provide background about concurrency platforms and system-call techniques in the following section. Section III presents our proposed architecture for concurrency platforms using asynchronous queue-based system-request techniques. We evaluate our architecture in Section IV, and discuss future research directions in Section V. Finally, we provide a conclusion in Section VI.

II. BACKGROUND

A. Concurrency Platforms

A concurrency platform enables the creation and execution of highly parallel applications. Every concurrency platform is composed of two components: a programming-language layer and a runtime system.

The goal of the programming-language layer is to reduce the developer’s burden of concurrent programming by providing language (or library) constructs to express concurrency in the application. These additional concurrency-control keywords may deliberately limit the freedom of concurrency the programming language allows to express. However, as a result of this, the number of pitfalls parallel programming involves is reduced, ultimately resulting in more robust applications. Additionally, the introduced limitation may be exploitable by the concurrency platform’s runtime system, increasing the execution efficiency [9]. Eventually, a concurrency platform’s runtime system is the essential driver of the parallel execution. Those runtime systems often employ randomized work-stealing as a scheduling strategy for scalability. This strategy assigns every “worker” a “ready-list” in the form of a queue. Workers only ever add new work to their assigned queue and they
only steal work from other worker’s queues if their associated queue runs out of work. Typically incarnations of workers are kernel-level threads, in the case of user-space runtime systems, or exclusively allocated cores in the case of spatially-multiplexing operating systems [10, 11]. Work-stealing is especially beneficial in the presence of many cores, as it effectively distributes the schedulers state, avoiding hot spots due to shared data.

We performed the experiments in this paper with our research concurrency platform named EMPER. The platform’s runtime system employs work-stealing to schedule fibers.

Figure 1 shows a concurrency platform running on top of an operating system. The compiler and the runtime system map the concurrency, expressed in the application’s source code, onto lightweight control-flow abstractions, called fibers. Figure 1 also demonstrates how a pseudo-blocking system call, which essentially looks like its synchronous blocking equivalent but does not suffer from the blocking anomaly, works internally. When the user invokes a pseudo-blocking system call, its request is wrapped in a future [12], which can be used to block until the request is completed. Those futures use synchronization-primitives native to the concurrency platform to possibly suspend the current execution entirely in userspace. As soon as a fiber (fiber #0 in Figure 1) must wait until a completion is available, the runtime system will dispatch another fiber (#4) on the worker thread, thus preventing the underutilization due to the blocking anomaly. The runtime system must fulfill the future’s promise in cooperation with the underlying operating system. In the following section, we present and discuss the suitability of available system call techniques to efficiently fulfill those promises.

### B. System-Call Techniques

The technique used by applications to submit requests to the operating system is essential for the application’s execution speed. The impact is, in particular, large if those applications are parallel and run on many cores. In the following, we will discuss different system-request techniques, present experimental results about their costs (shown in the “cost” column of Table I), and consider their suitability for many-core architectures.

#### 1) Synchronous And Blocking: The first category consists of synchronous blocking system calls, which are widely known and well understood. Monolithic kernels commonly use this technique. Every synchronous system call requires exactly one trap into the kernel for each request. This trap entails two context switches (denoted as “csw” in Table I), one into the kernel and one back into user-space. Those context switches can quickly become costly in terms of execution speed and energy consumption, especially when side-channel mitigations are enabled [14, 15]. Moreover, since those requests to the operating system may block the calling entity, they are the root cause of the blocking anomaly.

#### 2) Synchronous And Non-Blocking: In this regard preferable, variants are non-blocking synchronous system calls. Like their synchronous blocking counterparts, they require at least one trap into the kernel and two context switches. This minimum is achieved if the result of the system request is immediately available. However, if this is not the case, its satisfiability must be awaited via an I/O multiplexing facility like epoll(), and the request reissused. Using this facility imposes further traps into the kernel and context switches. Nevertheless, since multiple entities can be observed with a single system call, e.g. epoll(), the overhead due to the system call can be accounted to multiple operations, reducing the share of the trap/context switch of an individual operation.

Since it is guaranteed that the calling entity will not be blocked, the cause of the blocking anomaly is eliminated.

---

**TABLE I: Categories of system-call techniques**

<table>
<thead>
<tr>
<th>Kind</th>
<th>Mechanism</th>
<th>Examples</th>
<th>per sys request</th>
<th>cost[ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sync</td>
<td>Blocking</td>
<td>read(), write()</td>
<td>1</td>
<td>955 ± 1009</td>
</tr>
<tr>
<td>Sync</td>
<td>Non-Blocking</td>
<td>SOCK_NONBLOCK &amp; epoll()</td>
<td>[1, 3]</td>
<td>1656 ± 1318</td>
</tr>
<tr>
<td>Async</td>
<td>Callback</td>
<td>POSIX AIO</td>
<td>2</td>
<td>6224 ± 12232</td>
</tr>
<tr>
<td>Async</td>
<td>Queue-based</td>
<td>Linux AIO</td>
<td>3</td>
<td>1922 ± 1467</td>
</tr>
<tr>
<td>Async</td>
<td>Queue-based</td>
<td>io_uring</td>
<td>1</td>
<td>306 ± 636</td>
</tr>
</tbody>
</table>

**Figure 1:** The components of a concurrency platform. Cilk is used as example of the programming-language layer.
Kegel describes non-blocking system calls as an essential building block to solve the C10K problem [16], i.e., the problem of handling tens of thousands of connections by a single system. Eventually, this requires the application of the reactor design pattern, which is based on I/O event notification facilities like select() or epoll() and non-blocking I/O (using non-blocking system calls). Unfortunately, non-blocking I/O tends to be hard to implement. Furthermore, while there exist libraries, like libevent or libuv, that make the usage of non-blocking I/O easier, the resulting applications are still typically maintenance-intensive and challenging to debug. To make matters worse, a single request to the system can now require more than one trap into the operating system. For example, a read from a socket now involves the non-blocking read() invocation and potentially the blocking epoll() call plus the resubmission.

3) Asynchronous Callback-Based: We can bring the required traps per system request down again by using asynchronous callback-based system requests. Using those, only a single trap are needed to enter the kernel to submit the request. However, the response is not awaited right away. Instead, a subsystem will deliver the response, once it is available, via a callback.

POSIX AIO [13] is an example for such a system request interface. One callback mechanism specified by POSIX AIO is signal-based. However, signal-based callbacks have multiple drawbacks. First, the signal delivery may require an additional context switch and lowers the instructions-per-cycle (IPC) rate of the interrupted thread. Secondly, signals run asynchronously to the application. Hence, only functions explicitly declared as async-signal-safe by POSIX can be used within the signal handler, making them hard to use and error-prone. As an alternative, threads can be used to execute callbacks. Threads as callback carriers remedy some of the drawbacks of signal-based callbacks. However, threads may also require a context switch to start executing. Moreover, their main disadvantage is that they have a high ramp-up latency. For example, a typical Linux desktop system requires around 50µs to create and start executing a thread. Also, note that this time immensely varies, as it is influenced, among other things, by the operating system’s state at the time of thread creation. In addition, threads used to deliver callbacks compete with existing threads, and the unbounded creation of new threads could lead to an oversaturation of the system.

4) Asynchronous Queue-Based: Concurrency platforms can avert oversaturating the system by using asynchronous queue-based system requests. Here, the result is not delivered via a callback but placed into a queue, where the client can pick it up. This eliminates the requirement to create additional control flows. Further advantages over the currently widely used non-queue-based system-call mechanisms where demonstrated by Soares and Stumm with FlexSC [14]. The queue-based mechanism of FlexSC, where queues are used for submission and completion notification, allows for “exception-less system-calls”, meaning that no trap into the kernel is required to perform a system request. As a result, asynchronous queue-based techniques can avoid the overhead of switching the context and privilege level.

An example for an asynchronous queue-based system-call interface is Linux AIO\(^1\). However, Linux AIO never had a good standing within some parts of the Linux kernel developer community. In 2016 the lead developer of Linux, Linus Torvalds, characterized AIO as follows:

AIO is a horrible ad-hoc design, with the main excuse being "other, less gifted people, made that design, and we are implementing it for compatibility because database people — who seldom have any shred of taste — actually use it". But AIO was always really really ugly. (Torvalds, 2016 [17])

In 2019 an alternative to Linux AIO landed in form of io_uring [18] in Linux 5.1. Since then, io_uring is a candidate to become the generic asynchronous system-call interface of the Linux kernel. In the following, we briefly present the basic design of io_uring.

Figure 2 shows io_uring’s interface between kernel- and user-space. It consists of a pair of queues, which are implemented as ring buffers and named Submission Queue (SQ) and Completion Queue (CQ). Applications first obtain the next free Submission Queue Entry (SQE) out of the SQ, fill the SQE with the information required for their request and may notify the kernel about new SQEs. The kernel consumes those SEqs, processes them, and, upon completion, creates a Completion Queue Event (CQE) in the CQ. Applications may either periodically poll their CQ(s), or call io_uring_wait(), which is a library wrapper around io_uring_enter(), to wait for new completion events. The latter, io_uring_enter(), is the only potentially blocking system call of io_uring’s interface and of the same category as other I/O event multiplexers like select() / epoll().

\(^1\) Accessible via libaio (https://pagure.io/libaio), cf. the io(3) man page.
The designers of the io_uring interface thoughtfully considered hardware characteristics and exploited those where possible. For example, the size of a SQE is exactly 64 bytes, which matches the typical size of a cache line. Hence, the message-passing mechanism, supplied by the io_uring queues, exploits the parameters of the cache-coherency protocol of the underlying shared-memory architecture. Furthermore, since it is possible to submit multiple requests with a single trap, the individual share of the trap (and resulting context switches) of an operation can be below one if requests are batched.

Besides great performance due to the carefully designed interface io_uring provides additional features to further reduce an application’s system request overhead. Applications can submit requests to the system via io_uring without trapping into the kernel or requiring a context switch using a special “poller” kernel thread. This thread polls the SQs known to the kernel for new entries. If this mechanism is used, io_uring truly becomes a “exception-less system-call” mechanism. SQE requests can form a serialized chain of dependent operations, preventing the need of issuing a new request after a previous one finished as expected. Furthermore, frequently used resources, like file descriptors or buffers, can be registered with io_uring to reduce in kernel lookup overhead.

5) Cost comparison: We determined the cost of each technique under Linux 5.13 using a benchmark which performs a single read request from an event file descriptor on a 2.2 GHz Intel Xeon CPU (E5-4640). Since we ensured that the data was available at the time of the request, and given that event file descriptors are pure virtual objects, this setup mostly measures the overhead of the system-request technique.

The average execution times and their standard deviation are given in the “cost” column of Table I. Unsurprisingly, since it is a pure user-space implementation in glibc, POSIX AIO performs worst. Linux AIO yielded the second worst performance, which supports its bad standing within the kernel community. Not far behind was the epoll() based variant, which shows that, even though non-blocking approaches may be scalable, they still incur a relatively huge overhead. The second best execution performance was provided by the blocking system request approach. However, as discussed in Section I, it suffers from the blocking anomaly. By far the best performance was exhibited by io_uring.

### III. Architecture

The following shows how our architecture for pseudo-blocking system calls increases the efficiency of concurrency platforms, which offers the user an approachable API while using modern asynchronous queue-based system-request interfaces for performance and scalability.

Listing 3 presents a server equivalent to Listings 1 and 2. While this appears to be sequential code at the first look, a concurrency platform can serve multiple clients in parallel without facing the blocking anomaly, thanks to pseudo-blocking system calls. The function tcp_listener() is a functionality of the concurrency platform that spawns a fiber for every incoming connection on the provided TCP socket. The fiber will execute the given lambda function. Note all pretendedly blocking “system calls”, like read(2), are implemented as pseudo-blocking system calls.

The following sections present such an architecture and demonstrate our approach’s applicability via a case
study based on our research concurrency platform EMPER. However, since EMPER has many similarities with state-of-the-art runtime systems of concurrency platforms like Go, we assume that this architecture can be applied to those as well.

A. A Simple Architecture

To aid the understanding of the proposed architecture, we first discuss a simple architecture. Our final architecture can be seen as an extension of this simple architecture. This simple architecture of a concurrency platform interfacing with the operating system using asynchronous system calls can be seen in Figure 3. The basic building blocks of a work-stealing concurrency platform, the worker threads and their associated work-stealing queues (WSQ), are shown in the bottom right. The io_uring is wrapped in IoContext. Since io_uring’s SQ and CQ are not thread-safe, we ensure that the SQ is only ever accessed via a single control-flow by guarding its access via an actor [19] (aided by EMPER’s library support for actors). The created SubmitActor accepts new requests from the worker, creates SQEs for them and places them into the SQ. On the CQ end of the io_uring, a Completer thread receives the resulting CQEs and processes them. Typically a CQE causes a EMPER semaphore to be released, potentially unblocking a previously blocked fiber. Such unblocked, now runnable, fibers can usually not be put in any work-stealing queue due the asymmetry of their concurrently allowed operations: Only the queue-owning worker is allowed to enqueue new items into its queue. But workers are allowed to dequeue elements from other worker’s work-stealing queues. In return, it is possible to use lock-free queue algorithms [20].

If a runtime system like EMPER’s wants to make use of efficient non-blocking work-stealing queue algorithms, then a different approach is required to inject new work from outside the worker threads into the system. In EMPER we created a special-purpose queue for this, dubbed AnywhereQueue. From a scheduling point-of-view, this introduces multi-level queue scheduling [21], p. 196, where the AnywhereQueue resides at a level below the work-stealing queues. The AnywhereQueue is protected by a lock, since it is a multiple-producer multiple-consumer queue. This allows any queue operation from anywhere, hence the name.

While this architecture is simple, it has multiple drawbacks. First, the queues around the io_uring are also synchronized using locks. We already discussed the lock requirement for the AnywhereQueue. Additionally, the actor’s mailbox queue requires complex synchronization, involving mutexes. This allows the actor to be suspended once its mailbox becomes empty and to be resumed once new messages arrive in the mailbox. Another drawback of this architecture is, that the SubmitActor becomes a bottleneck that every io_uring operation has to pass. This is especially unfortunate, since we envision that in the future, concurrency platforms will exclusively use io_uring (or similar mechanisms) for every request to the operating system. Go’s netpoller architecture [22] follows this exact same design, only instead of io_uring Go uses epoll(), and hence suffers from the same limitations.

B. A Distributed and Scalable Architecture

We now present a distributed and scalable architecture based on the previous. The new architecture, as shown in Figure 4, includes two key improvements. First, instead of a single io_uring, every worker is assigned its own io_uring. Secondly, the Completer thread is now only used as fallback in case a worker does not check its CQ in a timely manner. This functionality is based on io_uring’s capability to monitor other CQs for new CQEs.

Using an io_uring per worker has multiple advantages: First, access to the SQ no longer needs to be synchronized, as only ever the worker adds new entries in the SQ. Secondly, the CQEs, which the worker obtained from its CQ, benefit from temporal and spatial locality.
Temporal locality is likely because io_uring is able to process SQEs inline if this is possible. An inline completed SQE causes the resulting CQE to be available right after the SQE was submitted. And since workers first submit new SQEs and then check the CQ for new CQEs, an immediately created CQE will be processed by the same worker within a short period of time. Spatial locality is provided by the fact that CQEs only ever appear in the io_uring of the corresponding SQ, i.e. the SQ where the SQE that caused the CQE was entered. Hence workers only ever reap the CQEs resulting from their own SQEs, and the action resulting from a CQE is often the continuation at the point where its SQE was submitted. Therefore, due to the spatial locality, the continuation after the pseudo-blocking system call likely benefits from hot CPU caches when being executed.

New work, in the form of CQEs, may get unnecessarily delayed in case the associated worker is currently executing a long-running fiber. To counter this, all worker io_uring are monitored for new CQEs by a “Global IO” io_uring, which is waited by the Completer thread. Once a new CQE in a worker’s io_uring appears, two threads may race to the related CQ: the worker and the completer. Hence, just like in the first simple architecture, the CQ needs to be synchronized by a lock. However, the performance penalty introduced by the worker-io_uring lock is far less than the penalty of the locks in the first architecture, because it is only ever taken by two threads. Our first preliminary experiments show that around 90% of the CQEs are consumed by the worker, and only a minor fraction is handled by the Completer.

Workers of a concurrency platform are always in an endless dispatch loop, independently of the used scheduling strategy (like work-stealing). They first obtain their next work item, in the form of a fiber, from the scheduler and then execute it. The presented architecture slightly modifies the dispatch loop: Workers check their CQ for new CQEs as part of the loop. This modification is generally applicable, independently of the used concurrency platform and scheduling strategy.

IV. Evaluation

In the following evaluation, we compare our proposed architecture against state-of-the-art production-grade concurrency platforms. Since the main use-cases of pseudo-blocking system calls involve (network) I/O, we first show a TCP echo benchmark as evaluation scenario. This class of benchmarks is common in evaluating network code and runtime systems [23]. The benchmark consists of an echo client, which measures the performance characteristics of the server implementation. We created three incarnations of the echo server, each using either EMPER, Go or Rust’s Tokio as concurrency platform, but otherwise try to reassemble a common code structure as much as possible.

We used an Intel Xeon system featuring 40 cores and 80 hardware threads across four NUMA nodes as server hardware. The client is a similar system featuring an Intel Xeon with 48 cores and 96 hardware threads also split between four NUMA nodes. Both systems are running Debian Buster with Linux 5.12 and are connected with 1Gbit/s NICs to the same switch with an average round-trip time of 0.15 ms.

The actual benchmark consists of two phases: During the first phase, all connections are established. Afterwards, in the second phase, the client sends 16-byte messages on each connection and expects to receive the same message back. The duration until all connections issued and received 2500 echos is used as the evaluation result. All presented line plots feature error bars showing the standard deviation of the experiment runs. Each TCP echo server implementation was compiled to native byte code using either Clang 11.1.0, Go 1.11.6, or Rust 1.48.0 and Tokio 1.9.0.

Figure 5 shows echo request-response throughput if the echo server uses all 80-cores. EMPER improves performance by 4% compared to Go. Both reach their peak throughput at 5000 connections, then show a decrease...
in the overall throughput as the number of connection increases. While Tokio can provide better throughput if fewer connections are used, it peaks at 1000 connections and cannot reach the throughput levels of EMPER and Go. Using 500 or fewer connections causes the throughput to vary heavily. We attribute this to the different worker suspension strategies of the runtime systems. A more aggressive suspension strategy will result in increased power savings, traded for decreased throughput. We assume the reason that the throughput of all runtime systems declines if more than 5000 connections are used lies outside of the runtime systems themselves, e.g., in the kernel. A non-runtime-system specific reason would explain why all three runtime systems are affected so similarly.

In this first benchmark the echo server does not perform any computational task when performing the echo request. While this can be considered a worst-case scenario for the involved runtime and operating system and provide valuable insights, it is not a realistic workload. Figure 6 shows the previous experiment with the same parameters but with an additional computational part on the server-side when processing the echo request. We introduced an artificial computation of 200 µs by busy waiting before sending the response. Most characteristics of the performance curves from the previous experiment without computation reappear here too. The throughput peak is still achieved by EMPER at 5000 connections, and after that, the throughput decreases. Due to the added computational part, Go and Tokio switched places. While Go can keep the throughput nearly constant up to 25 000 connections, EMPER achieves 25.1% more throughput.

To determine the impact of a runtime system (and the used operating system request interface), we reduce the number of worker threads the echo server uses to 20. We also restrict those 20 workers to the 20 cores of the system’s first NUMA node, to which the network interface is attached. As in the previous experiment, the echo server is configured with an 200µs computation overhead. By applying Little’s law $L = \lambda W$ [24], we can determine the maximum possible throughput by resolving it to the arrival rate $\lambda$. Using 20 workers ($L = 20$) performing $W = 200\mu s$ work, yields Equation (1).

\[
\lambda = \frac{L}{W} = \frac{20}{200\mu s} = 0.1 \frac{1}{\mu s} = 100 000 \frac{1}{s} \quad (1)
\]

Hence, according to Little’s law, the maximum throughput possible is 100 000 echos per second. Note that this is under the simplification that the work-span consists only of the computation. However, in the real system, the work-span also includes, amongst other things, scheduling within the runtime-system and in-kernel I/O processing.
Figure 7: Simultaneous client connections plotted against echo throughput of an echo server using 20 worker threads (one NUMA node) with 200 µs computation per echo based on EMPER, Go, and Rust Tokio.

As shown in Figure 7, EMPER can achieve more throughput, around 3.6 %, better than the best competitor (Tokio). Compared to Go, EMPER achieves 10.1 % more throughput. And we have to keep in mind that, Go and Tokio are production-grade concurrency platforms, whereas EMPER is a research prototype. It is clear to us that EMPER’s runtime system can be further improved, and the real performance gap between the used system request technique uncovered.

The used concurrency platform affects the applications service quality. To present this effect, Figure 8 shows the 95th, 99th, 99.99th and, 99.999th percentiles of the response times as observed by the client for 10 000 connections and without restrictions to the worker count or artificial computation. As can be seen, the response times within up to the 99th percentile of EMPER and Go are comparable. At the 99th percentile, Tokio’s services quality degenerates, exhibiting increased response times. The response times increase for all systems at the 99.99th and 99.999th. At the 99.999th percentile, EMPER has the lowest latency. It is 7.7 % lower compared to Go, and 76.7 % compared to Tokio.

To not only explore a single kind of IO we also developed and evaluated a simple file-content search benchmark. The benchmark recursively searches all files in the Linux 5.13 source tree for the needle `io_uring`.

The Go and EMPER implementations concurrently walk the directory tree, read all regular files in 4K chunks and search the retrieved memory for the needle. In addition to the concurrency platform we measured the specialized search tool ripgrep [25] without file filtering or regular expressions to obtain comparable results. The presented data was collected on the same 48 core Intel Xeon machine the echo client was running on with the Linux source tree residing on a NFS file system.
Figure 10: Echo throughput using 10,000 simultaneous client connections of EMPER echo servers running on different Linux kernel versions

Figure 9 presents the results of fifteen runs of the file-content search benchmark. EMPER achieves $1.17 \times$ faster search times than the other concurrent platform Go. And all concurrency platforms outperform the file-content search tool ripgrep using a thread pool and blocking system calls.

A different somewhat meta effect is that io_uring is a relatively new Linux kernel feature. As io_uring attracts more and more users, making it more and more important, it is constantly being optimized. Hence it seems reasonable to expect further io_uring-internal improvements leading to “free” performance boosts in user programs employing this interface. Figure 10 presents the same echo server experiment running on different kernel versions. It shows a trend to increased performance with newer Linux kernel versions. The fact that the io_uring version featured in Linux 5.12 achieves the highest throughput supports our claims about the nature of this actively developed and researched interface. The end of the line has not yet been reached.

Those observations, the additional features mentions in Section II-B4, and the high possibility that io_uring will become the general asynchronous system-call interface of the Linux kernel makes us confident in our interface choice and that there are and will be more space worth exploring in the context of concurrency platforms.

V. Future Work

Using a Completer thread demonstrably helps reducing (tail) latency. But it is also a necessity of the presented architecture because the runtime system’s asynchronous system-call interface and the worker suspension mechanisms are decoupled. The Completer prevents possible sleep locks where IO completions arrive at the io_uring of a suspended worker. This calls for an io_uring based worker-suspension mechanism.

For example, if io_uring would support futexes [26], then those could be used for worker-suspension aware of available system-request completions. A worker would suspend itself waiting on its CQ when observing no executable work in the system. As soon as the kernel generates a new CQE, be it as completion notification of a system request or because another worker posts a notification about new work, the worker would be resumed. However, futexes cannot precisely notify one of many workers due to the limited size of FUTEX_WAKE_BITSET. Furthermore, futexes do not provide a way to pass additional information, to hint where new work is available.

Hence we argue, that this calls for a kernel-level solution tailored for the worker-suspension use case. Such a solution would consist of workers sleeping in io_uring_wait_cqe(), woken up through a CQE which potentially contains further information about the wakeup cause, e.g. a pointer to the queue where new work is available. The kernel would track the number of sleeping workers and make that information available to user space via shared memory between the kernel-space and user-space. Workers producing new work, e.g. the awakening site of the scheme, would check this counter, and if it is non-zero, issue an SQE instructing the kernel to wake another worker. We expect the “sleeping workers” counter to be mostly zero for sufficiently parallel applications.

One of the most promising techniques we are currently investigating is I/O stealing. Our design, where every worker has a dedicated I/O completion queue (CQ), already reassembles the structure of a work-stealing scheduler, just for I/O completions. Workers then would also try to steal I/O completions from other workers’ CQs. And since the design of io_uring makes it possible to reap the I/O completions in a lock-free manner, this should not have a heavy performance impact. Our first preliminary experiments with I/O stealing support this assumption and show encouraging results.

For the future, we assume that the ongoing trend
Towards concurrency platforms has significant consequences for operating-system kernel design. Figure 11 lifts the curtain behind the abstract kernel “cloud” in Figure 2 and shows some of the internal machinery that io_uring consists of. Note how multiple kernel worker threads are used to process the io_uring requests. Those workers consume their jobs from a dedicated queue. The whole picture closely resembles a concurrency platform, just in kernel space.

While the io_uring API is remarkably easy to use, the in-kernel machinery required to drive it is very complex. This complexity has multiple reasons.

First, the machinery needs to perform accounting of in-flight requests and resource usage to prevent applications from being able to flood the kernel with SQEs that never (or at least not within a reasonable timeframe) complete. This alone is already quite challenging, and the io_uring developers just finished a solution.2

Secondly, the inherent concurrent nature of the in-kernel machinery requires proper synchronization of the shared data structures. Fortunately, the Linux kernel provides a wide range of synchronization primitives and mechanisms, ranging from modern locks to mechanisms like read-copy-update [27]. However, the application of those is often non-trivial and requires thorough consideration.

This poses new challenges to kernel development. We have to ask ourselves if we are building a concurrency platform within the kernel. Moreover, if the answer is yes, which seems pretty likely, then the question is not if we should entangle the user-space and kernel-space concurrency platforms, but how this should be done. An example where such an entanglement is beneficial revolves around the various kernel-level threads involved in the operation of a concurrency platform. It is easy to overlook that there are not only the worker threads that the runtime system uses to execute fibers. In the case of io_uring, the kernel also creates I/O threads. Those I/O threads are regular kernel-level user threads, processing the io_uring requests within the kernel, and hence the main difference from threads created from user space is that they never return to user space. So while concurrency platforms try to avoid overprovisioning the system with threads by restricting their number to the available cores, the additional I/O threads, created as a side-effect of using io_uring, are now causing such an overprovision. Preventing this requires close cooperation of the kernel space and user-space concurrency platforms and is part of our future work.

Considering that many-core architectures are a reality today and the high probability that the number of cores will increase further, we conclude that systems-software research is more relevant than ever.

VI. CONCLUSION

In this paper, we propose the concept of pseudo-blocking system calls via asynchronous queue-based system-request techniques. Based on its implementation and evaluation, we demonstrate its unique advantages in direct comparison with today’s state-of-the-art in concurrency platforms (i.e., Google’s Go and Rust’s Tokio). We propose the adaptation of pseudo-blocking system calls as it remains an unsolved challenge to write efficient parallel code that is easy to understand and maintain. With pseudo-blocking system calls, programmers are no longer required to resort to complex event-based designs for handling I/O intensive workloads in parallel. Instead, program code can be structured using traditional blocking functions without being affected by blocking anomalies at runtime. We further present architectural adjustments of concurrency platforms to efficiently use modern system-call techniques, enabling scalable pseudo-blocking system calls. The results of our evaluation show that the evaluated prototype already yields competitive performance. The peak throughput of an echo server implementation was improved by 4% on a 80-core machine compared to the second-best contender, while tail latency was reduced by 7.7%.
In a file-content search benchmark, our approach outperformed the second-best contender by $1.17 \times$. These results are very encouraging, considering that this is an early prototype implementation contesting with state-of-the-art production-grade concurrency platforms. Finally, we made the case that the kernel-space and user-space concurrency platforms need to be entangled for optimal efficiency. Since the related kernel subsystem (i.e., io_uring) is relatively new, we are confident that its future improvements will further increase the advantage of our architecture.

Open Source

EMPER, the used applications and evaluation framework are open source and available under:
- gitlab.cs.fau.de/i4/manycore/emper
- gitlab.cs.fau.de/i4/manycore/emper-io-eval
- gitlab.cs.fau.de/i4/manycore/emper-fs-eval
- gitlab.cs.fau.de/i4/manycore/syscall-eval

REFERENCES


