technische universität dortmund

fakultät für informatik

Programmable Asynchronous I/O with io_uring and eBPF in Linux

Franz–B. Tuneke*, Christian Dietrich+, Horst Schirmeier*

Herbsttreffen 2021 der Fachgruppe Betriebssysteme

Lehrstuhl für Informatik 12 (*)
TU Dortmund

technische universität dortmund

Operating System Group (+)
TU Hamburg

TUHH

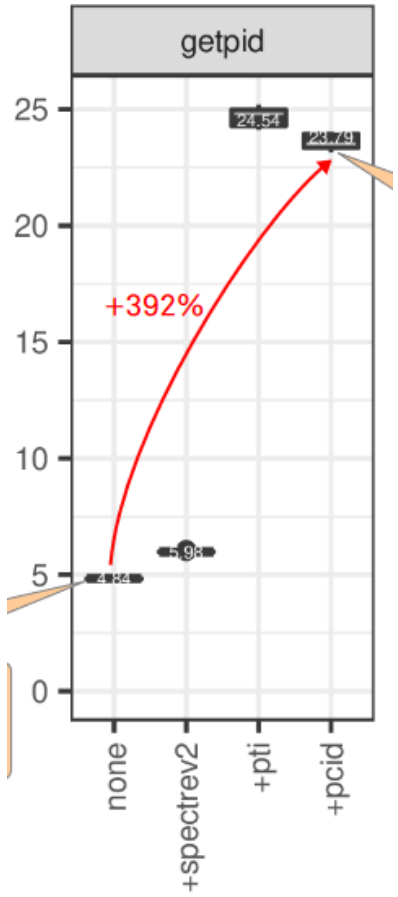
Agenda

1. Motivation
2. io_uring als Sammelschnittstelle
3. eBPF-basiertes Programmiermodell
4. Anwendungsfall: Key-Value-Store Mrcache
5. Fazit

Motivation

- Systemaufrufe besitzen Overhead durch Moduswechsel
  → Bis zu 20 mal mehr Overhead als „normale“ Funktionsaufrufe [6]
- Vergrößert durch Maßnahmen gegen Spectre/Meltdown
  → Im Extremfall bis zu 400 % erhöhte Laufzeit [5]
- Blockweise Kopieren: Blockgröße 4 KB, Dateigröße 100 MB
  → systemspezifische 23,391 ms Overhead durch Moduswechsel (ca. 18 % der Gesamtzeit)

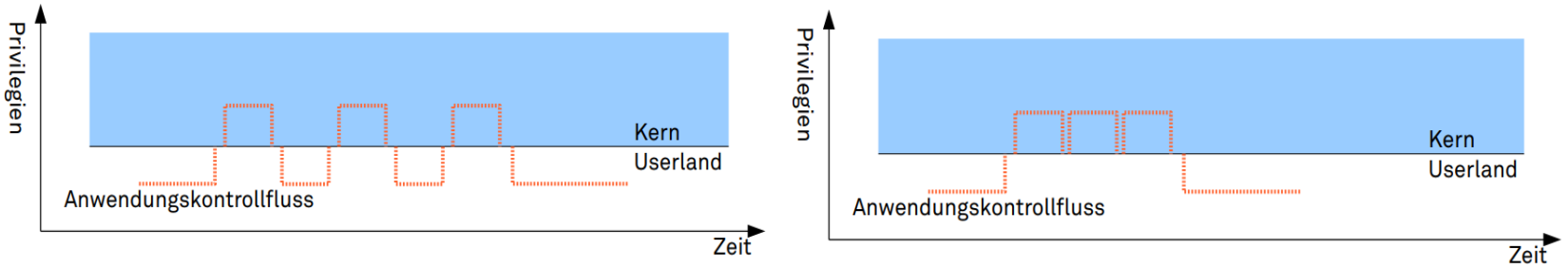
Quelle: [5]



[5] H. Schirmeier. 2018. *Revisiting OS Multi-Calls in the Light of Meltdown and KPTI*. url: https://ess.cs.tu-dortmund.de/downloads/syscall-aggregation-talk.pdf
[6] J. Mauro. 2000. Solaris Internals Core Kernel Components.

Motivation

- Verschiedene Ansätze „Kosten“ zu reduzieren
  - Neue anwendungsfallorientierte Systemaufrufe (*sendfile()*) [2]
  - Synthese spezialisierter leichtgewichtiger Systemaufrufe [3]
  - Gesammeltes Ausführen von Systemaufrufen („Sammelschnittstelle“) [1, 2, 4, 7]



Quelle: [5]

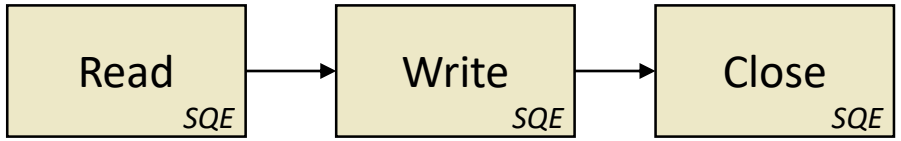
[1] L. Soares, M. Stumm. „FlexSC: Flexible System Call Scheduling with Exception-Less System Calls.“ In: *Osdi*. Bd. 10. 2010, S. 1
[2] E. Zadok u. a. „Efficient and safe execution of user-level code in the kernel“. 2005. doi: 10.1109/IPDPS.2005.189.
[3] C. Pu, H. Massalin und J. Ioannidis. „The synthesis kernel“. In: *Computing Systems* 1.1 (1988), S. 11–32
[4] M. Rajagopalan u. a. „Cassyopia: Compiler Assisted System Optimization.“ In: *HotOS*. Bd. 3. 2003, S. 1–5.
[5] H. Schirmeier.. *Revisiting OS Multi-Calls in the Light of Meltdown and KPTI*. 2018.
[7] A. Koomshin, Y. Shinjo. Running application specific kernel code by a just-in-time compiler. 2015.

Agenda

1. Motivation
2. io_uring als Sammelschnittstelle
3. eBPF-basiertes Programmiermodell
4. Anwendungsfall: Key-Value-Store Mrcache
5. Fazit

io_uring als Sammelschnittstelle

- API für asynchrone Ein-/Ausgabe
- Systemaufrufe werden als SQEs im Kernelmodus ausgeführt



- SQEs erzeugen CQEs bei Fertigstellung
- *io_uring* als Sammelschnittstelle Gegenstand von Studienarbeit [8]
  → Analyse von Systemaufrufmustern in klassischen Unix-Werkzeugen

io_uring – cat mit bekannter Eingabelänge

- Anzahl an Systemaufrufen berechenbar

```
nr_of_read_writes = FILE_SIZE / BLOCKSIZE;
for(int i = 0; i < nr_of_read_writes; i++){
    ret = read(fd1, buf, BLOCKSIZE);
    if (ret > 0){
        if (write(fd2, buf, BLOCKSIZE) <= 0)
            return -1;
    }
    else
        return -1;
}
//<Hier analog dazu Rest schreiben>
```

io_uring – cat mit bekannter Eingabelänge

- Anzahl an Systemaufrufen berechenbar

```
nr_of_read_writes = FILE_SIZE / BLOCKSIZE;
for(int i = 0; i < nr_of_read_writes; i++){
    ret = read(fd1, buf, BLOCKSIZE);
    if (ret > 0){
        if (write(fd2, buf, BLOCKSIZE) <= 0)
            return -1;
    }
    else
        return -1;
}
//<Hier analog dazu Rest schreiben>
```

io_uring – cat mit bekannter Eingabelänge

- Anzahl an Systemaufrufen berechenbar
- Rückgabewerte nur für Fehlerbehandlung
  ➔ Keine Abhängigkeiten

```
nr_of_read_writes = FILE_SIZE / BLOCKSIZE;
for(int i = 0; i < nr_of_read_writes; i++){
    ret = read(fd1, buf, BLOCKSIZE);
    if (ret > 0){
        if (write(fd2, buf, BLOCKSIZE) <= 0)
            return -1;
    }
    else
        return -1;
}
//<Hier analog dazu Rest schreiben>
```

io_uring – cat mit bekannter Eingabelänge

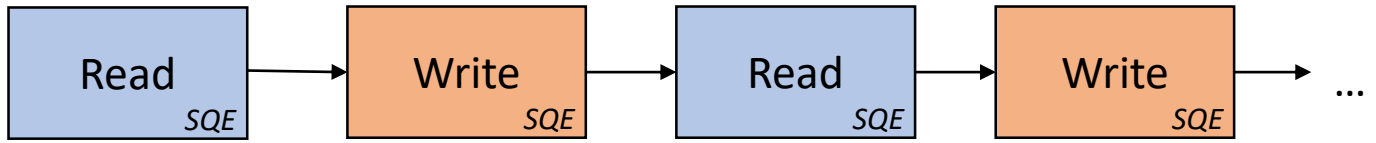
- Anzahl an Systemaufrufen berechenbar
- Rückgabewerte nur für Fehlerbehandlung
  ➔ Keine Abhängigkeiten

```
nr_of_read_writes = FILE_SIZE / BLOCKSIZE;
for(int i = 0; i < nr_of_read_writes; i++){
    ret = read(fd1, buf, BLOCKSIZE);
    if (ret > 0){
        if (write(fd2, buf, BLOCKSIZE) <= 0)
            return -1;
    }
    else
        return -1;
}
//<Hier analog dazu Rest schreiben>
```

io_uring – cat mit bekannter Eingabelänge

- Anzahl an Systemaufrufen berechenbar
- Rückgabewerte nur für Fehlerbehandlung
  ➔ Keine Abhängigkeiten
- Mit *io_uring* nur zwei Moduswechsel

```
nr_of_read_writes = FILE_SIZE / BLOCKSIZE;
for(int i = 0; i < nr_of_read_writes; i++){
    prep_linked_read_sqe(fd1, buf, BLOCKSIZE);
    prep_linked_write_sqe(fd2, buf, BLOCKSIZE);
}
//<Hier analog dazu Rest behandeln>
io_uring_submit_and_wait();
```



io_uring – cat mit unbekannter Eingabelänge

- Anzahl an read-/write-Aufrufen vorab unbekannt

```
do{
    count = read(fd1, buf, BLOCKSIZE);
    if (count > 0){
        if(write(fd2, buf, count) <= 0)
            return -1;
    }
    else if (count < 0)
        return -1;
}while(count != 0);
```

io_uring – cat mit unbekannter Eingabelänge

- Anzahl an read-/write-Aufrufen vorab unbekannt

```
do{
    count = read(fd1, buf, BLOCKSIZE);
    if (count > 0){
        if(write(fd2, buf, count) <= 0)
            return -1;
    }
    else if (count < 0)
        return -1;
}while(count != 0);
```

io_uring – cat mit unbekannter Eingabelänge

- Anzahl an read-/write-Aufrufen vorab unbekannt
- Anzahl gelesener Bytes durch read vorab unbekannt
  ➔ Wie viel mit SQEs schreiben?
- Abhängigkeit zwischen read/write

```
do{
    count = read(fd1, buf, BLOCKSIZE);
    if (count > 0){
        if(write(fd2, buf, count) <= 0)
            return -1;
    }
    else if (count < 0)
        return -1;
}while(count != 0);
```

io_uring – cat mit unbekannter Eingabelänge

- Anzahl an read-/write-Aufrufen vorab unbekannt
  ➔ Wie viele SQEs?
- Anzahl gelesener Bytes durch read vorab unbekannt
  ➔ Wie viel mit SQEs schreiben?
- Abhängigkeit zwischen read/write

```
do{
    count = read(fd1, buf, BLOCKSIZE);
    if (count > 0){
        if(write(fd2, buf, count) <= 0)
            return -1;
    }
    else if (count < 0)
        return -1;
}while(count != 0);
```

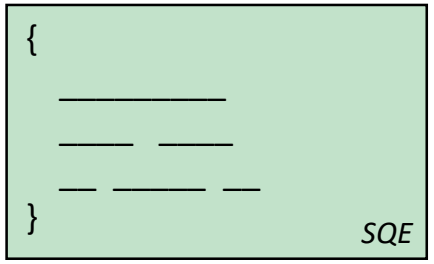
- Mehrere Moduswechsel auch mit *io_uring* nötig, je nach Eingabelänge
- Bei komplexeren Mustern ist Sammeln nicht möglich

Agenda

1. Motivation
2. io_uring als Sammelschnittstelle
3. eBPF-basiertes Programmiermodell
4. Anwendungsfall: Key-Value-Store Mrcache
5. Fazit

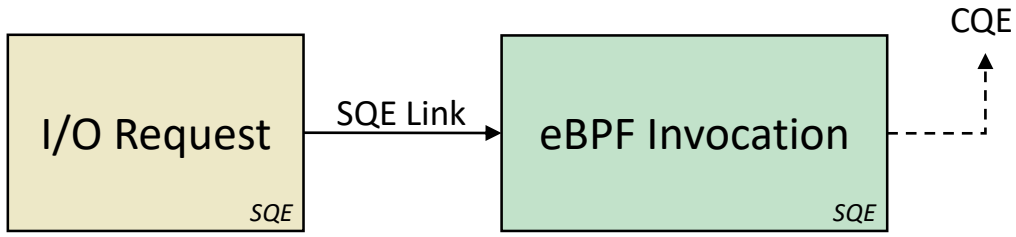
eBPF-basiertes Programmiermodell

- Benutzerdefinierter Quellcode durch VM im Kernel ausführbar
- Restriktionen aus Sicherheitsgründen: Bounded Loops, 512 B Stack, …
- Kommunikation mit Userspace über Maps möglich
- Erweiterung von io_uring um eBPF
  → Neue SQE-Klasse: eBPF-SQEs

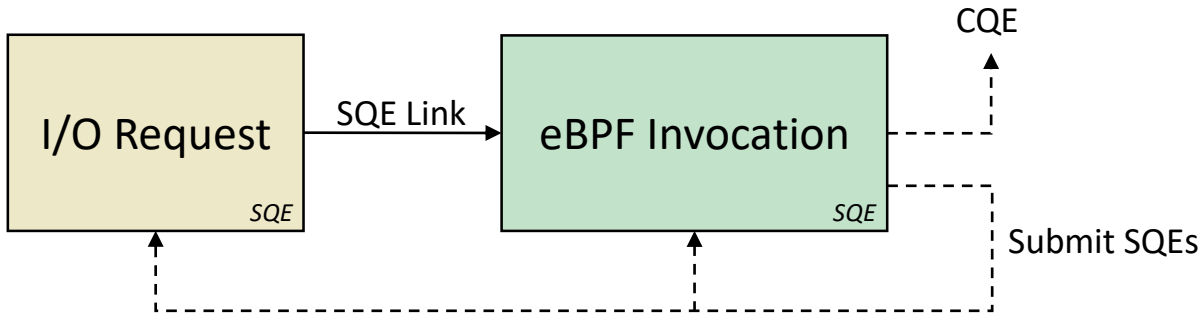


eBPF-basiertes Programmiermodell

- Beispiel „Postprocess“



- Beispiel „Loop“



Agenda

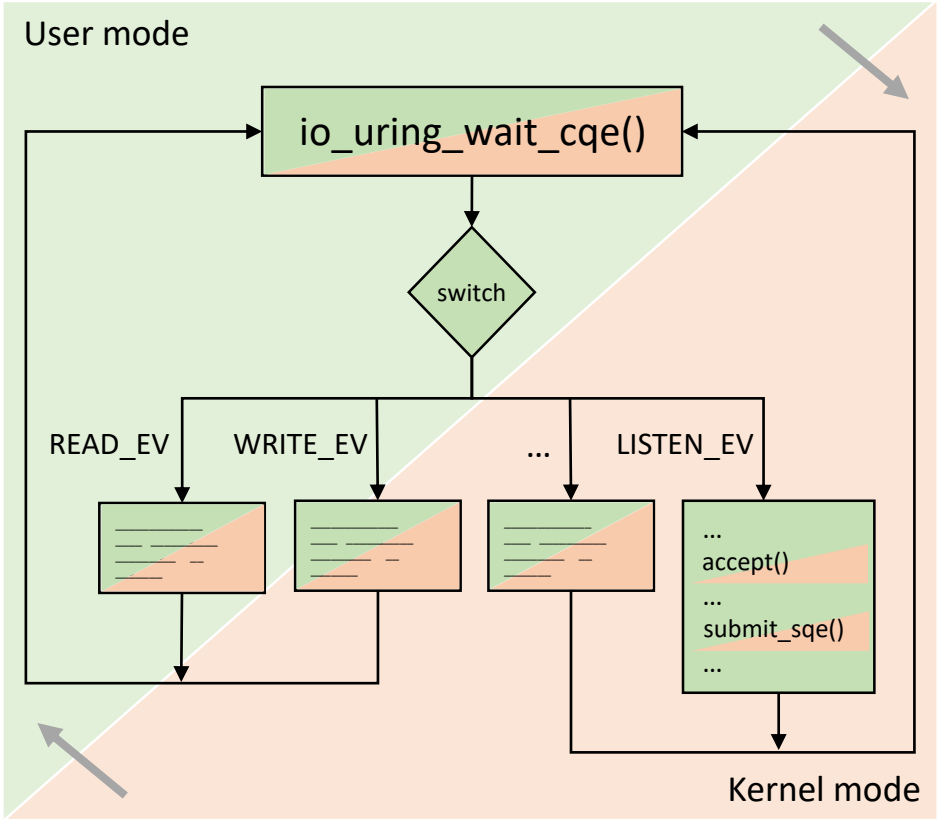
1. Motivation
2. io_uring als Sammelschnittstelle
3. eBPF-basiertes Programmiermodell
4. Anwendungsfall: Key-Value-Store Mrcache
5. Fazit

Anwendungsfall: Key-Value-Store Mrcache

- Key-Value-Store in C (https://github.com/MarkReedZ/mrcache)
  - Verwendet *io_uring* in Vanilla-Version
  - Benutzt „normale“ Systemaufrufe (accept, write, …) + SQEs
    → Viele Moduswechsel
- Unsere Version
  - Verwendet *io_uring* + eBPF-Erweiterung
  - Initialisierung + Speicherallokierung in Userspace
  - Get-Funktionalität komplett im Kernelspace
  - Set-Funktionalität noch nicht implementiert

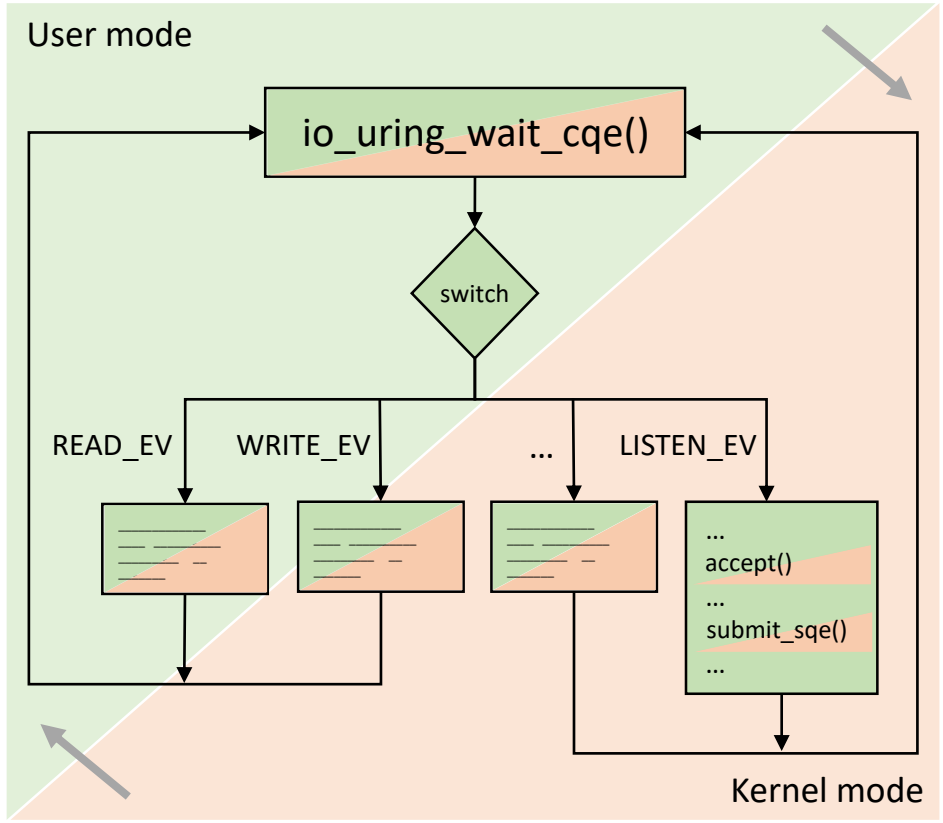
Anwendungsfall: Key-Value-Store Mrcache

Originalversion: Event Loop mit io_uring

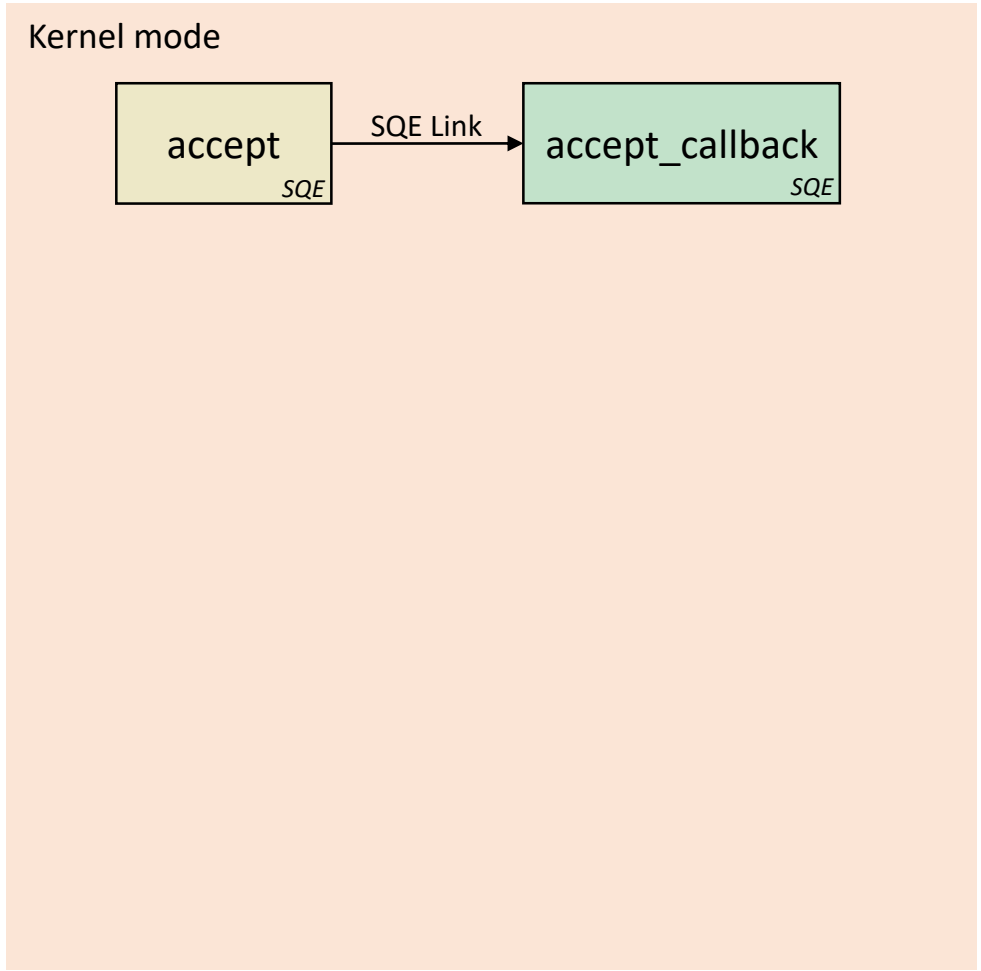


Anwendungsfall: Key-Value-Store Mrcache

Originalversion: Event Loop mit io_uring

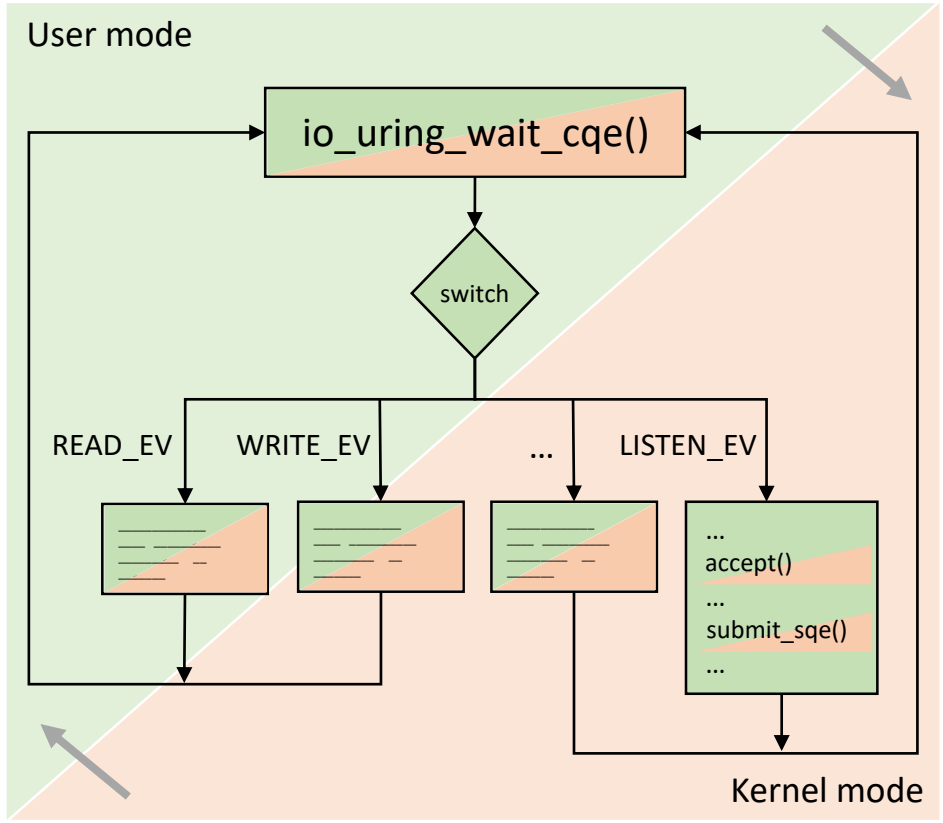


Neue Version: io_uring + eBPF

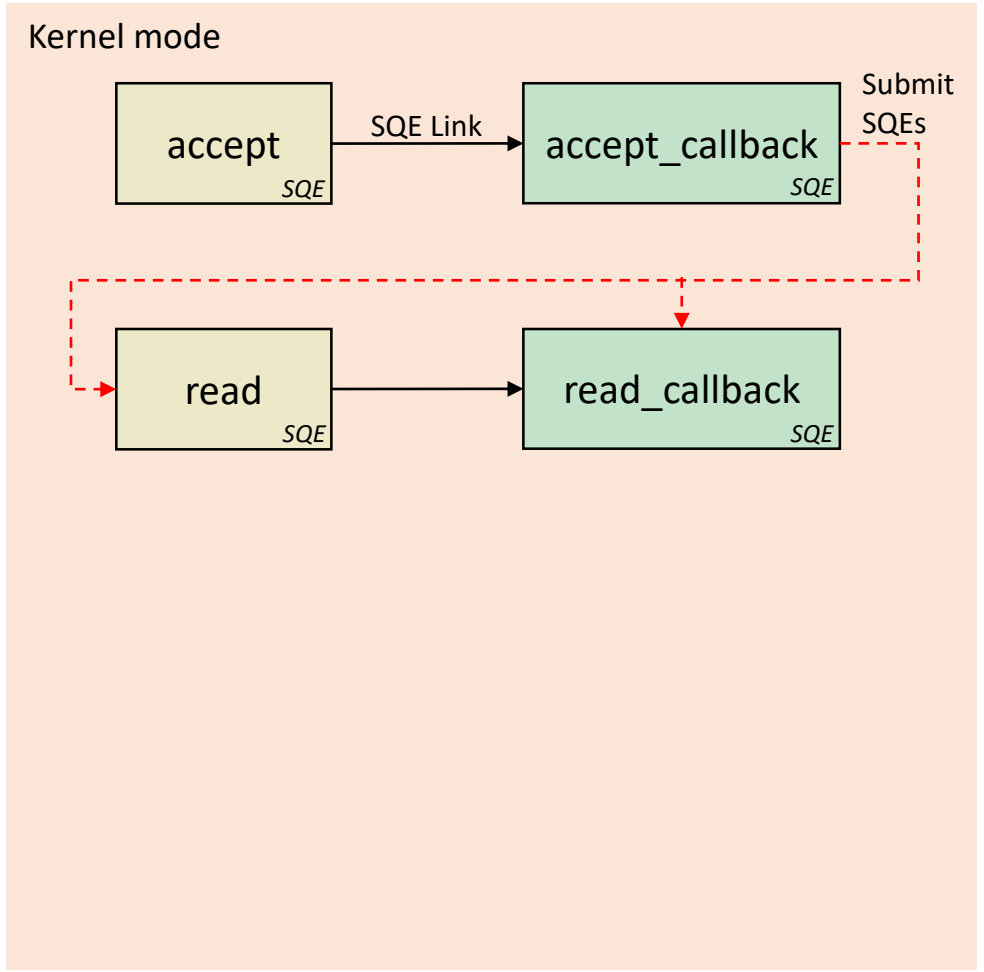


Anwendungsfall: Key-Value-Store Mrcache

Originalversion: Event Loop mit io_uring

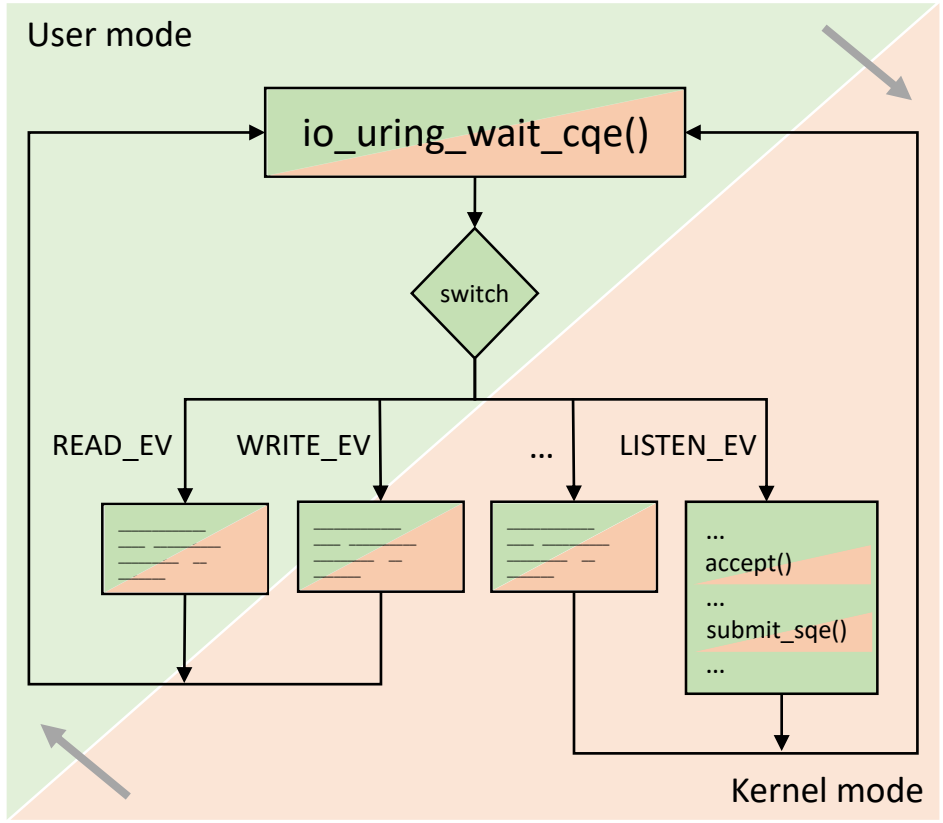


Neue Version: io_uring + eBPF

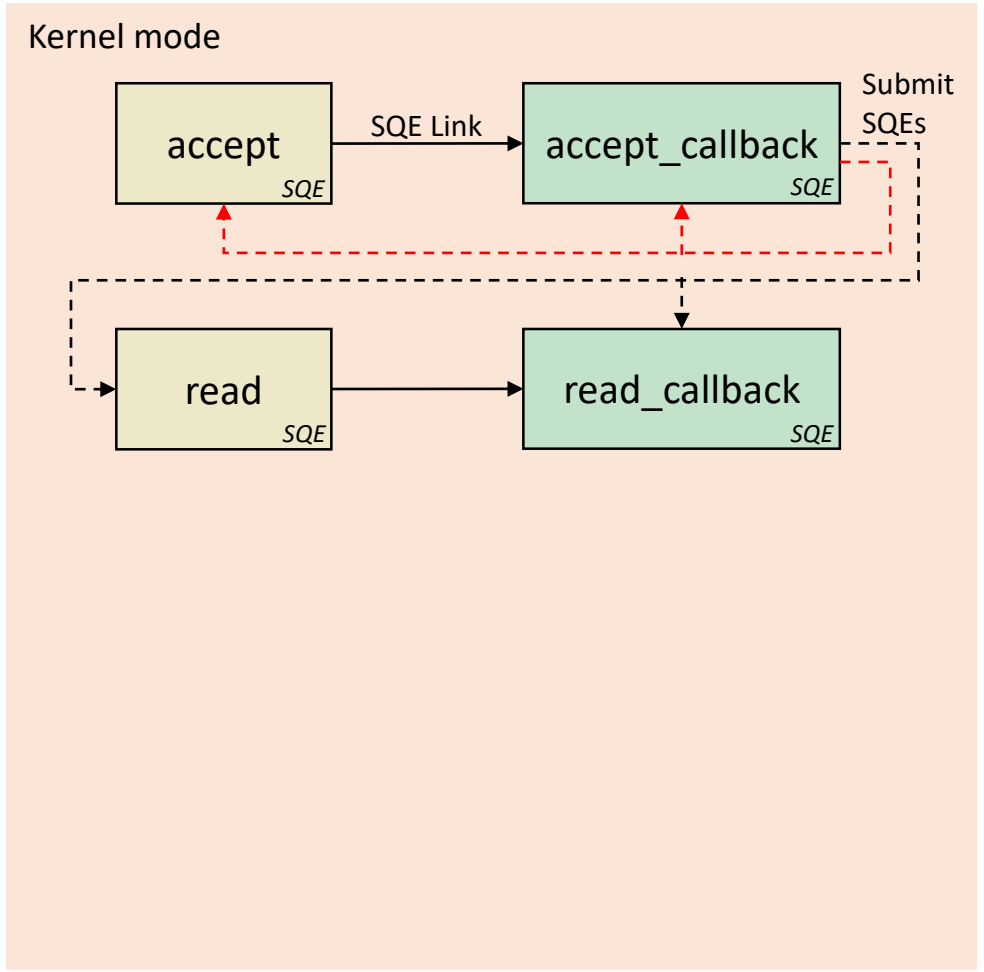


Anwendungsfall: Key-Value-Store Mrcache

Originalversion: Event Loop mit io_uring

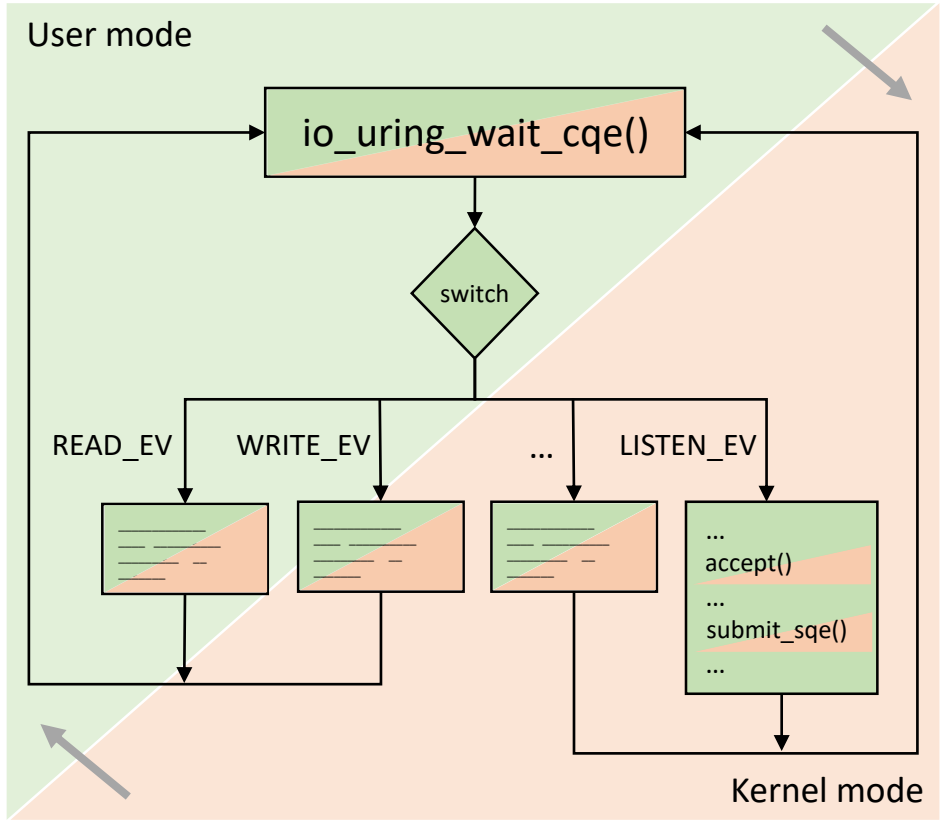


Neue Version: io_uring + eBPF

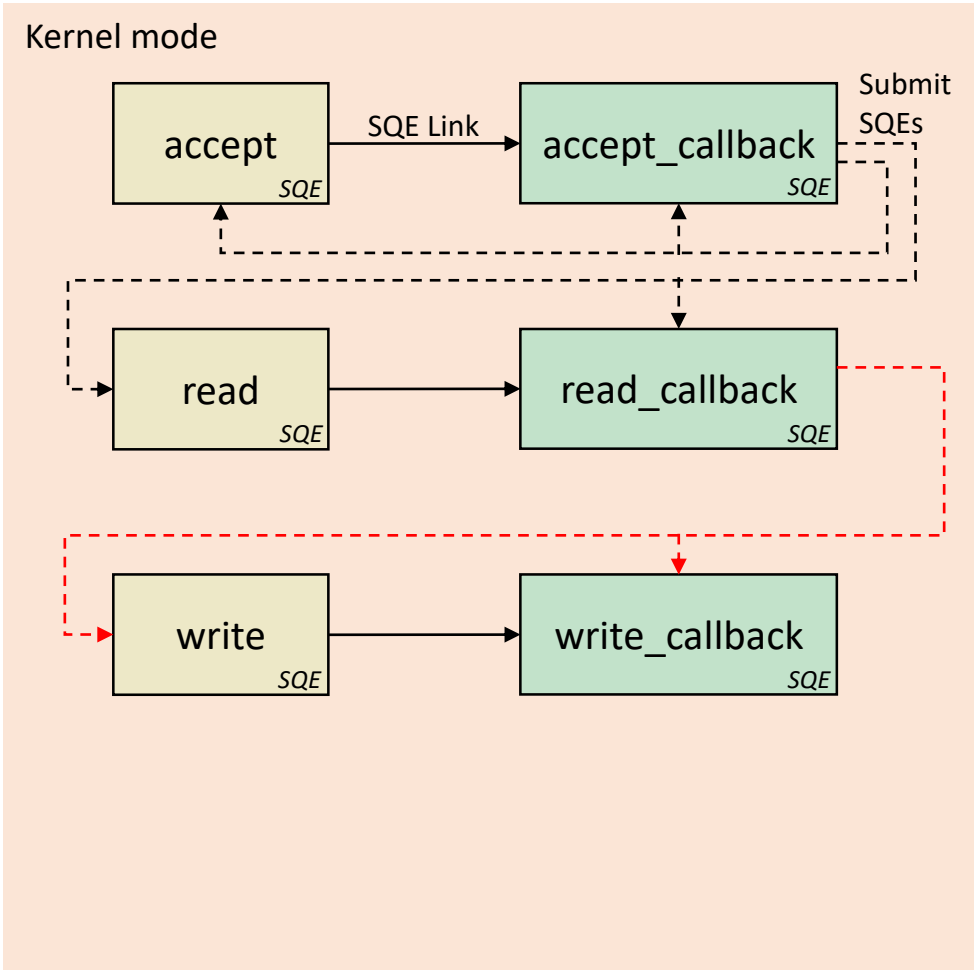


Anwendungsfall: Key-Value-Store Mrcache

Originalversion: Event Loop mit io_uring

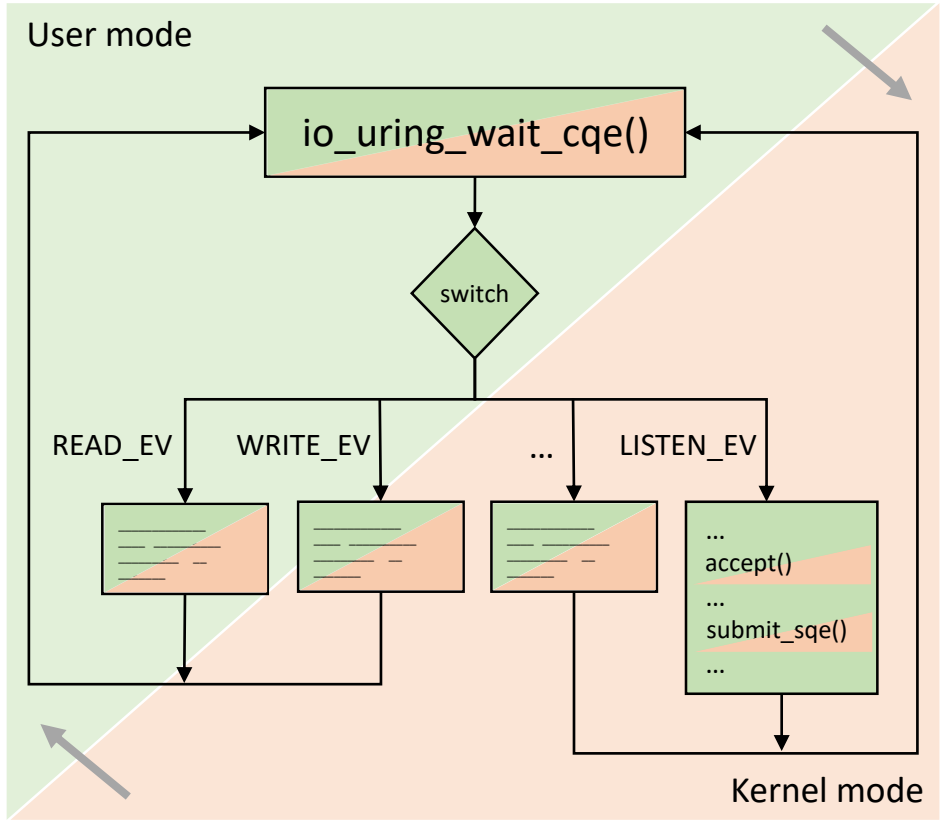


Neue Version: io_uring + eBPF

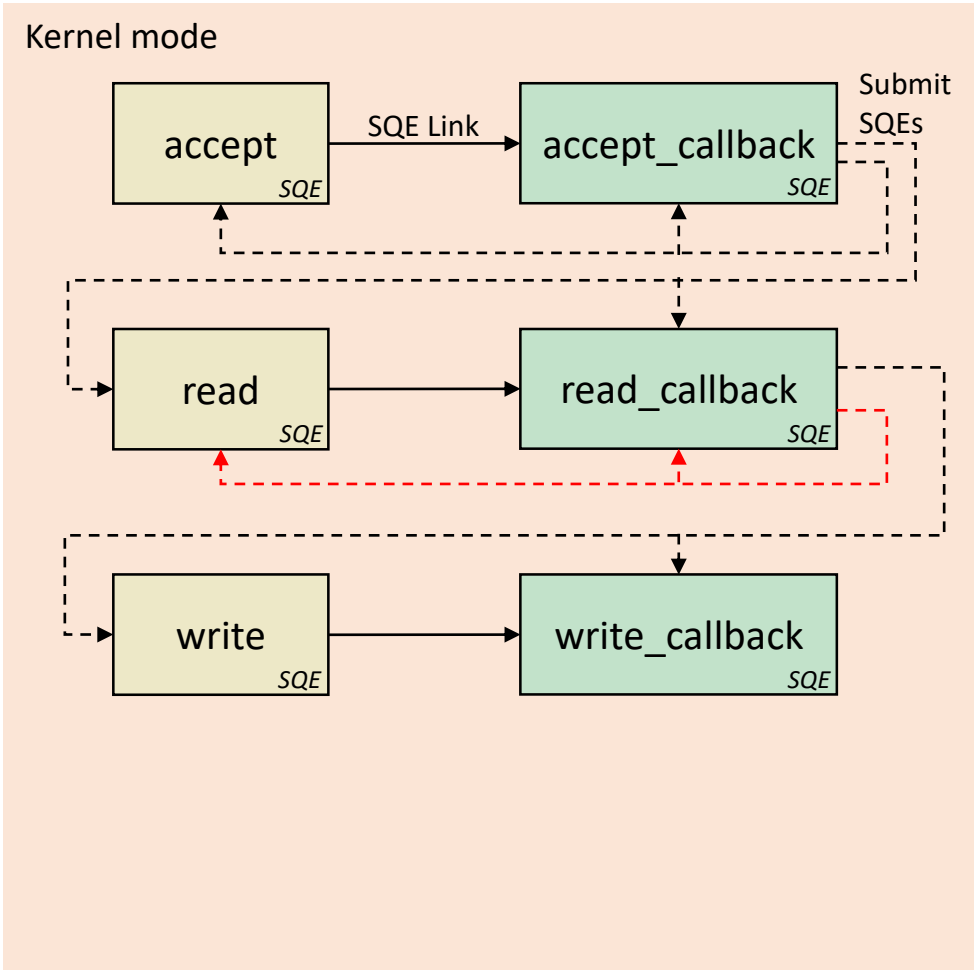


Anwendungsfall: Key-Value-Store Mrcache

Originalversion: Event Loop mit io_uring

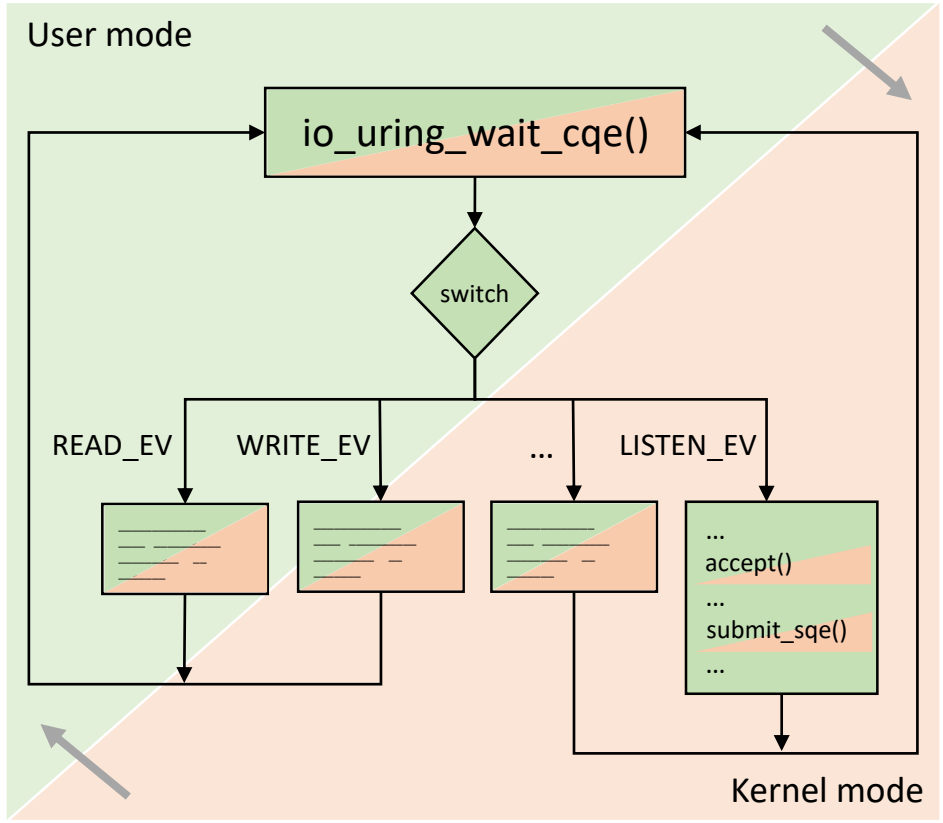


Neue Version: io_uring + eBPF

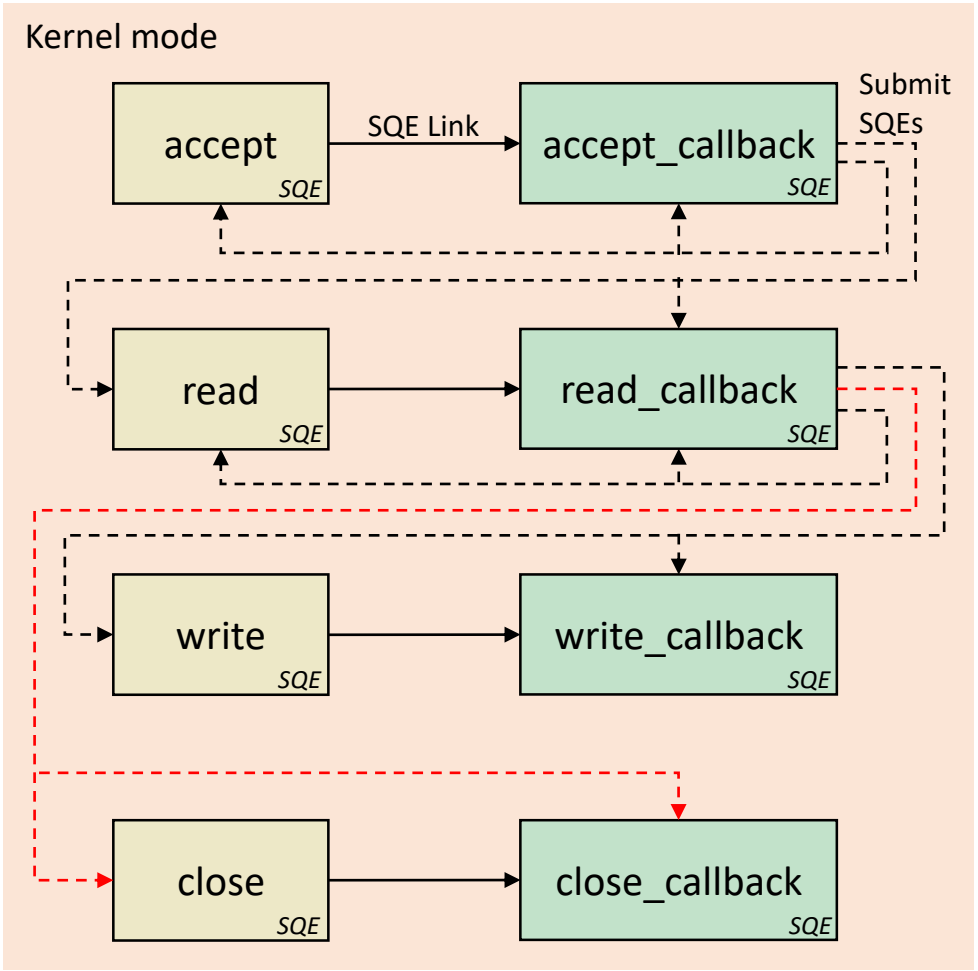


Anwendungsfall: Key-Value-Store Mrcache

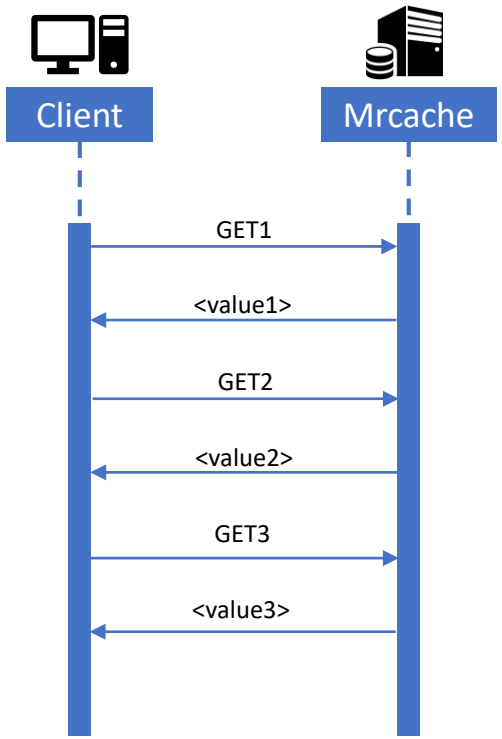
Originalversion: Event Loop mit io_uring

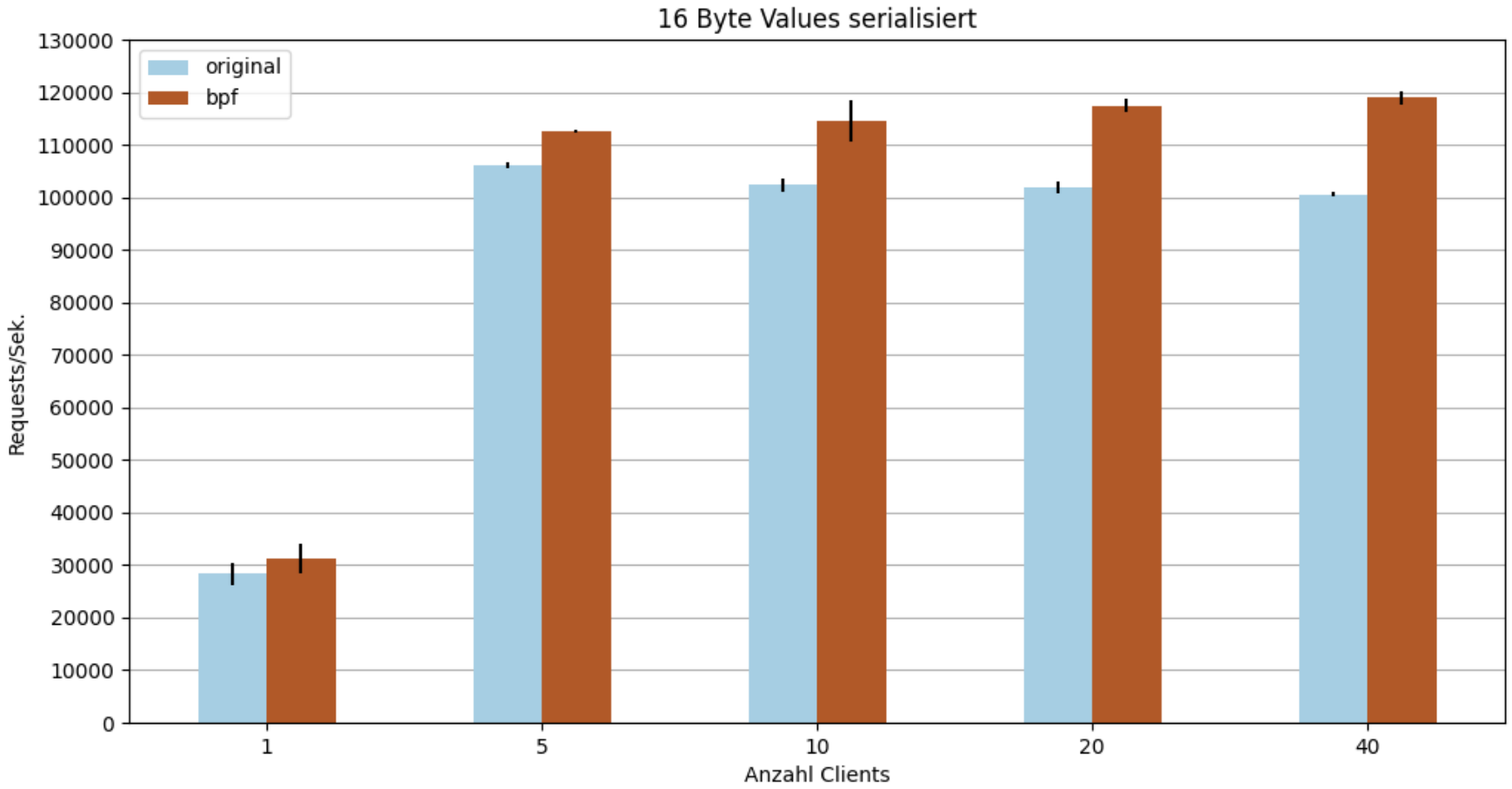


Neue Version: io_uring + eBPF

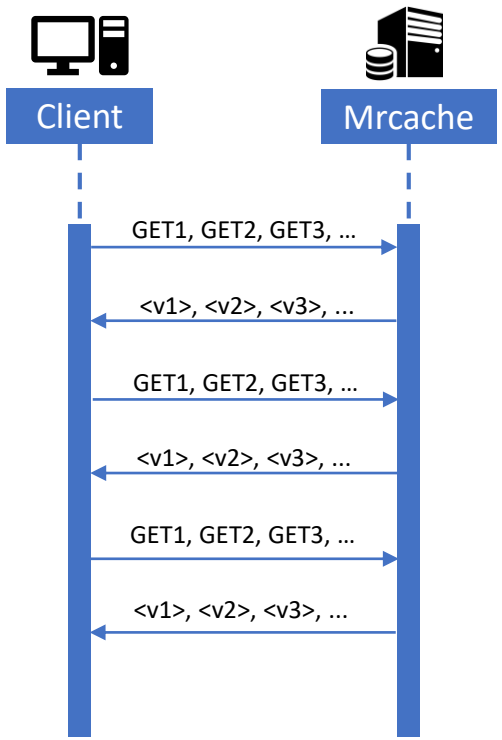


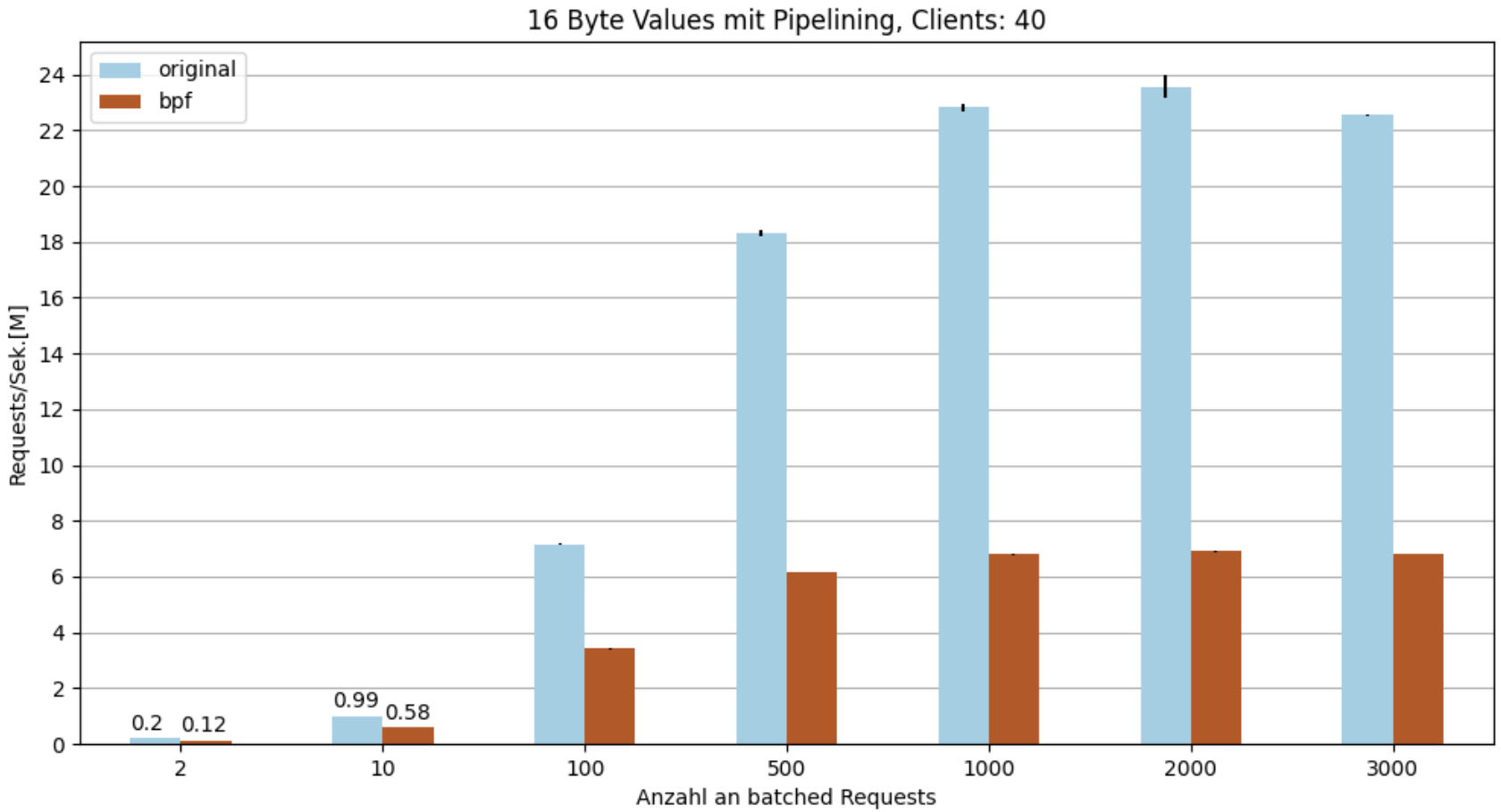
Messergebnisse Single Requests



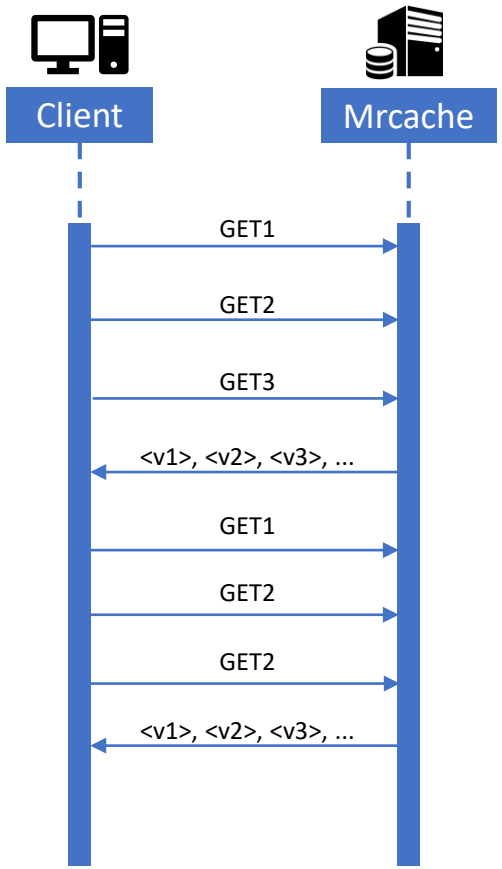


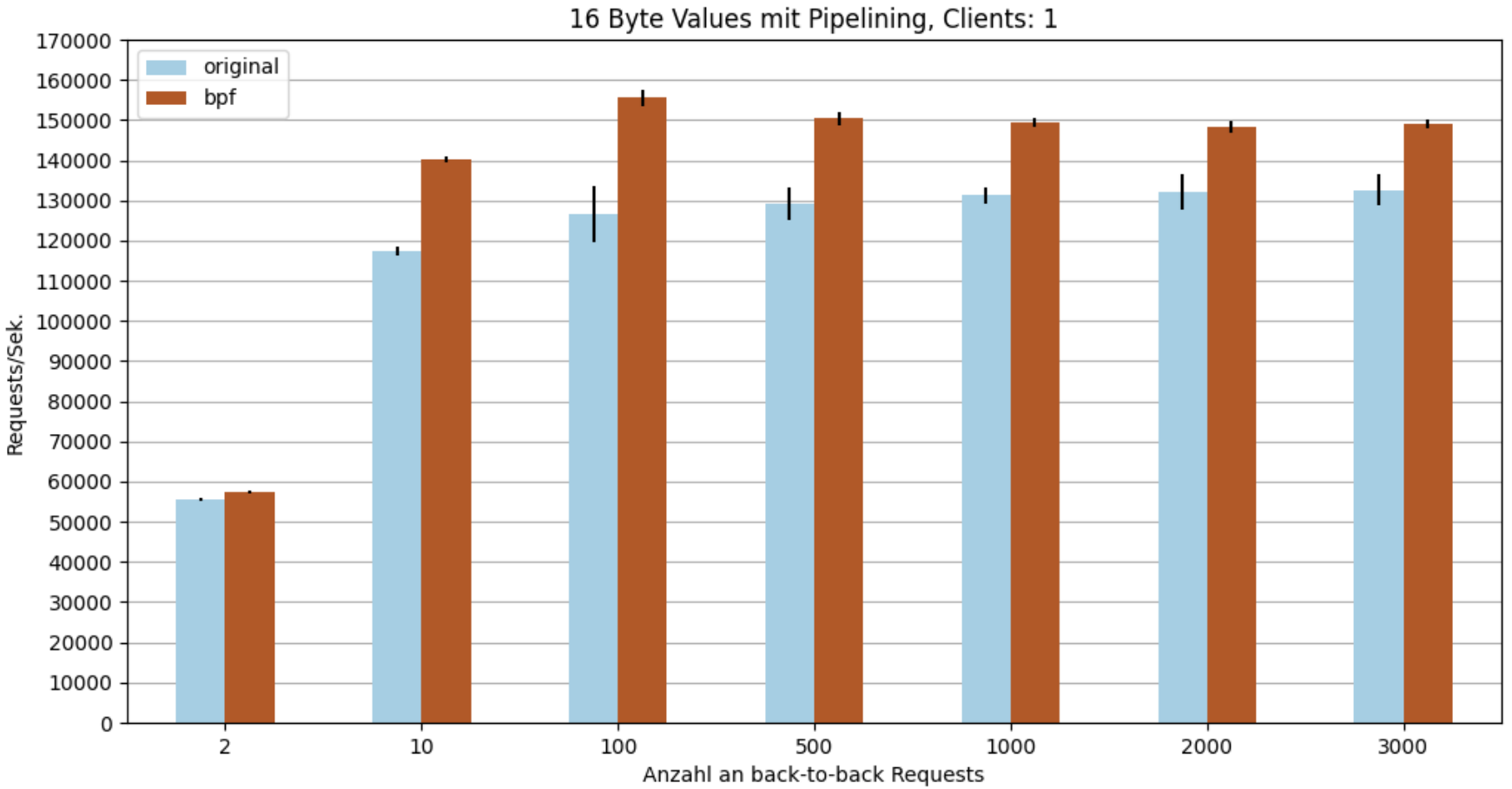
Messergebnisse Pipelining – Batched





Messergebnisse Pipelining – Back-2-Back





Agenda

1. Motivation
2. io_uring als Sammelschnittstelle
3. eBPF-basiertes Programmiermodell
4. Anwendungsfall: Key-Value-Store Mrcache
5. Fazit

Fazit

- Systemaufrufe durch Spectre/Meltdown deutlich „teurer“ geworden
  - Erster Ansatz: Systemaufrufe sammeln für weniger Moduswechsel
  - Abhängigkeiten problematisch → *io_uring* mit eBPF-Unterstützung
- Anwendungsfall „Mrcache“ zeigt vorhandenes Potential
  - Höherer Durchsatz bei einzelnen Get-Requests
  - Niedrigerer Durchsatz bei Batching
  - Teilweise höherer Durchsatz bei Back-2-Back-Requests
- Ausblick: Weiteres Potential laut *io_uring*-Experten vorhanden
  - Overhead Reduzieren
  - Parallele Ausführung von eBPF-SQEs → Synchronisierung?

Quellen

[1] L. Soares und M. Stumm. „FlexSC: Flexible System Call Scheduling with Exception-Less System Calls.“ In: *Osdi*. Bd. 10. 2010, S. 1–8

[2] E. Zadok u. a. „Efficient and safe execution of user-level code in the kernel“. In: *19th IEEE International Parallel and Distributed Processing Symposium*. 2005, 8 pp.-. doi: 10.1109/IPDPS.2005.189.

[3] C. Pu, H. Massalin und J. Ioannidis. „The synthesis kernel“. In: *Computing Systems* 1.1 (1988), S. 11–32

[4] M. Rajagopalan u. a. „Cassyopia: Compiler Assisted System Optimization.“ In: *HotOS*. Bd. 3. 2003, S. 1–5.

Quellen

[5] H. Schirmeier. 2018. *Revisiting OS Multi-Calls in the Light of Meltdown and KPTI*. url: https://ess.cs.tu-dortmund.de/downloads/syscall-aggregation-talk.pdf (besucht am 18.05.2021)

[6] J. Mauro. 2000. *Solaris Internals Core Kernel Components*.

[7] KOOMSIN, Ake; SHINJO, Yasushi. Running application specific kernel code by a just-in-time compiler. In: Proceedings of the 8th Workshop on Programming Languages and Operating Systems. 2015. S. 15-20

[8] Franz-B. Tuneke. 2020. Analyse von Systemaufrufmustern für POSIX-Sammelschnittstellen