ORWC - open read write close

open

Die Latenz einer open-Operation wurde mit dem Program *filesystem.c* in 1.000.000 Durchläufen gemessen und in eine .csv Datei ausgegeben.
Um Latenzen, die durch das Verwenden verschiedener Kerne entstehen, zu eliminieren wurde der Prozess an einen festen Kern gebunden:

```
static void pin_to_cpu(int cpu){
    cpu_set_t set;
    CPU_ZERO(&set);
    CPU_SET(cpu, &set);
    if(sched_setaffinity(0, sizeof(set), &set) != 0){
        perror("sched_setaffinity");
    }
}
```

Darüber hinaus findet ein WarmUp statt, um zusätzliche Latenzen bei der ersten Aufrufen zu eliminieren, die z.B. durch das Caching von Instruktionen und Datenstrukturen im Kern entstehen:

```
// warmup
for(int i = 0; i < 100; i++){
    int fd_null = open("/dev/null", O_RDWR);
    if(fd_null < 0){
        perror("open /dev/null");
        free(buf);
        return 1;
    }
    close(fd_null);
}
```

Die eigentliche Zeitmessung verwendet schließlich *clock_gettime()* und schreibt das jeweilige Ergebnis anschließend in eine csv Datei:

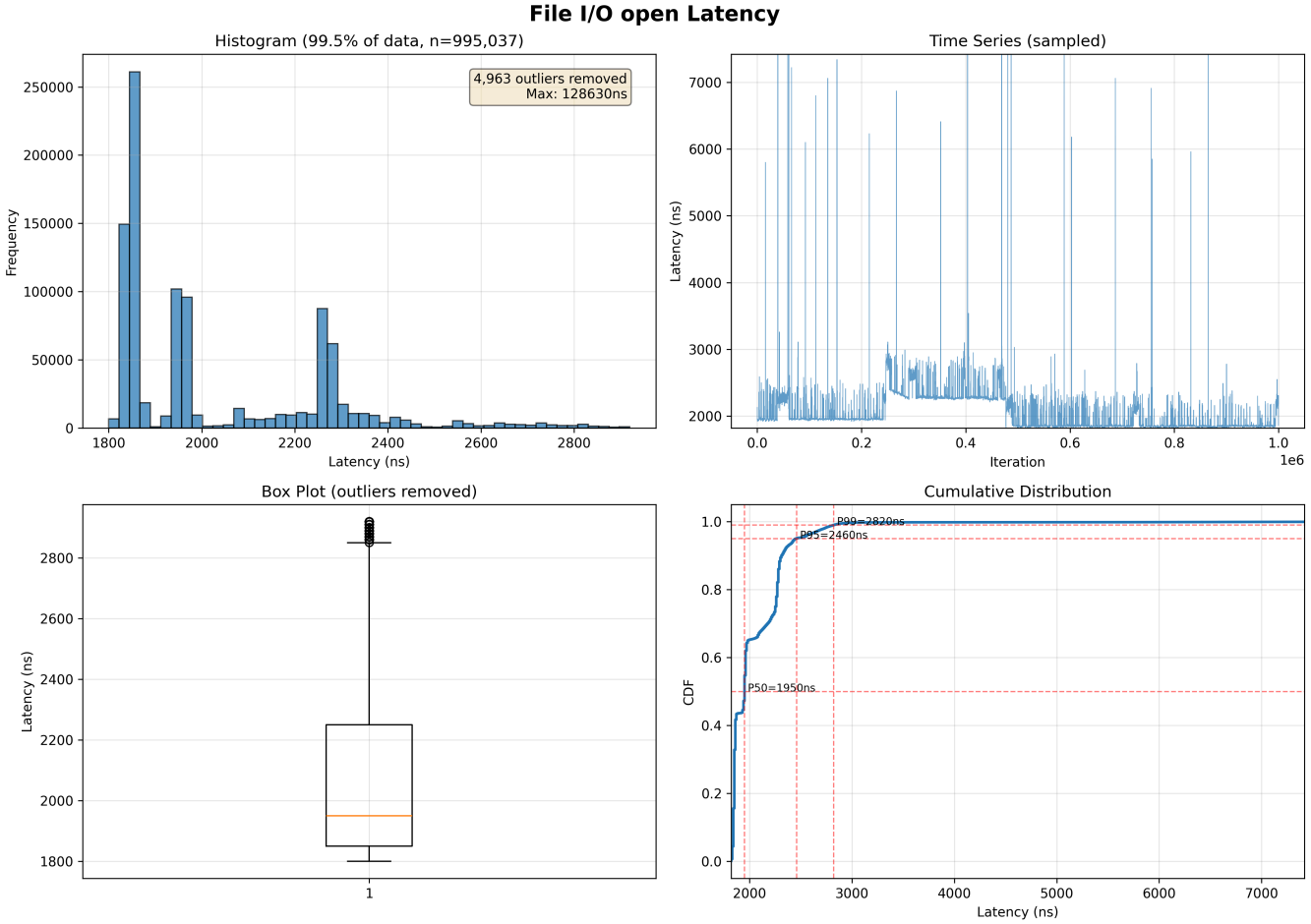
```
for(int i = 0; i < iters; i++){

    clock_gettime(CLOCK_MONOTONIC, &start);
    int fd_null = open("/dev/null", O_RDWR);
    if(fd_null < 0){
        perror("open /dev/null");
        free(buf);
        return 1;
    }
    clock_gettime(CLOCK_MONOTONIC, &end);
    double latency = timespec_diff(start, end);
    fprintf(csv_open, "%.2f\n", latency);
    ...
}
```

/

Die Funktion *timespec_diff()* zur Berechnung der Latenz in Nanosekunden wurde außerdem als *static inline* definiert:

```
static inline double timespec_diff(struct timespec start, struct timespec end){
    return (end.tv_sec - start.tv_sec) * 1e9 + (end.tv_nsec - start.tv_nsec);
}
```

Dabei wird der Overhead für den Eintritt und den Exit des Syscalls (Kontextwechsel), der Overhead der VFS (hier ext4) Layer und der Overhead der */dev/null* Treiberoperationen gemessen. Insbesondere fallen durch die Verwendung von */dev/null* keine Hardwarelatenzen an.
Die Messungen haben die folgenden Ergebnisse geliefert:



Count: 1,000,000
Mean: 2046.62 ns
Median: 1950.00 ns
Std Deviation: 390.02 ns
Min: 1800.00 ns
Max: 128630.00 ns
95% CI: [2045.85, 2047.38] ns
50th Percentile: 1950.00 ns
95th Percentile: 2460.00 ns
99th Percentile: 2820.00 ns

/

read()

Die Latenz einer Leseoperation wurde mit dem Program *filesystem.c* in 1.000.000 Durchläufen gemessen und in eine .csv Datei ausgegeben.
Auch für die Messung der Latenz der Leseoperation wurde der Prozess an einen festen Kern gebunden und ein WarmUp durchgeführt.

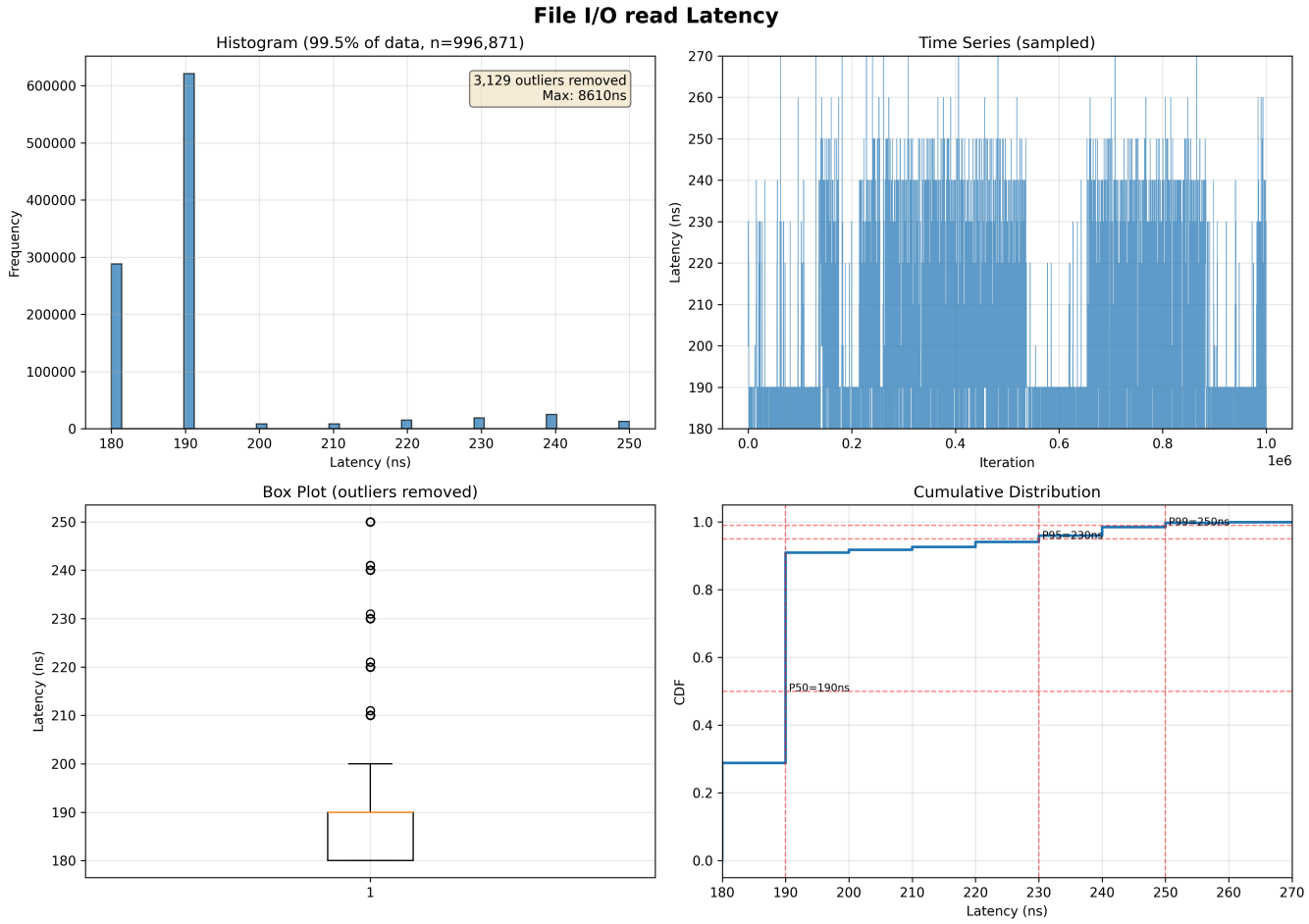
```
// warmup
for(int i = 0; i < 100; i++)
    read(fd_null, buf, block_size);
```

Es wurde ebenfalls *clock_gettime()* für die einzelnen Zeitmessungen verwendet und das jeweilige Ergebnis anschließend in eine csv Datei geschrieben:

```
for(int i = 0; i < iters; i++){
    // measuring writes
    clock_gettime(CLOCK_MONOTONIC, &start);
    read(fd_null, buf, block_size);
    clock_gettime(CLOCK_MONOTONIC, &end);
    double latency = timespec_diff(start, end);
    fprintf(csv_read, "%.2f\n", latency);
}
```

Die Funktion *timespec_diff()* ist ebenfalls als *static inline* definiert.
Auch hier wird der Overhead für den Eintritt und den Exit des Syscalls (Kontextwechsel), der Overhead für die VFS (hier ext4) Layer und der Overhead für die */dev/null* Treiberoperationen gemessen. Insbesondere fallen durch die Verwendung von */dev/null* keine Hardwarelatenzen an.
Die Messungen haben die folgenden Ergebnisse geliefert:

/



Count: 1,000,000
Mean: 191.70 ns
Median: 190.00 ns
Std Deviation: 64.29 ns
Min: 180.00 ns
Max: 8610.00 ns
95% CI: [191.57, 191.82] ns
50th Percentile: 190.00 ns
95th Percentile: 230.00 ns
99th Percentile: 250.00 ns

/

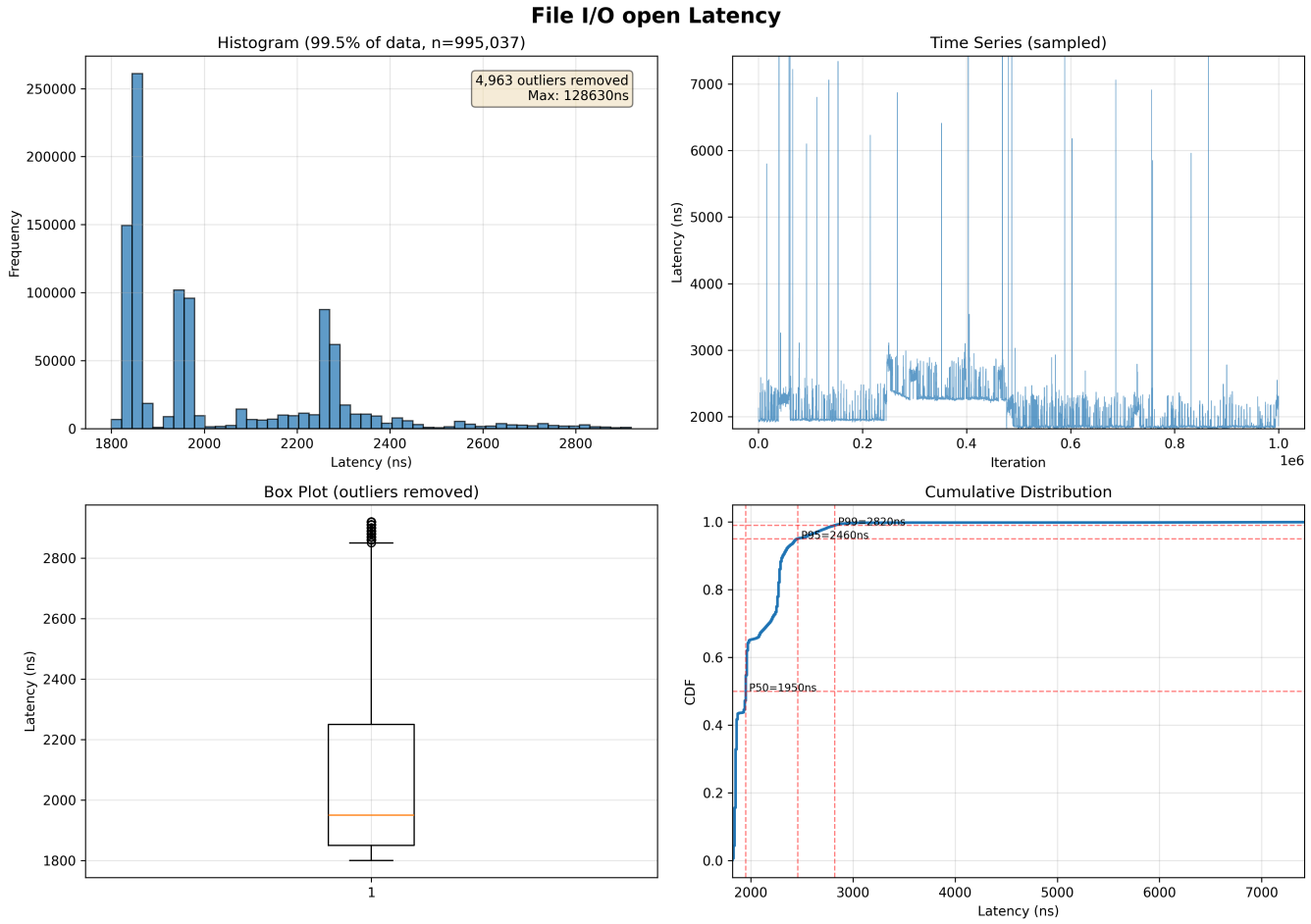
write()

Die Latenz einer Schreiboperation wurde mit dem Program *filesystem.c* in 1.000.000 Durchläufen gemessen und in eine .csv Datei ausgegeben.
Auch für die Messung der Latenz der Schreiboperation wurde der Prozess an einen festen Kern gebunden und ein WarmUp durchgeführt (vgl. read).
Es wurde ebenfalls *clock_gettime()* für die einzelnen Zeitmessungen verwendet und das jeweilige Ergebnis anschließend in eine csv Datei geschrieben:

```
for(int i = 0; i < iters; i++){
    // measuring writes
    clock_gettime(CLOCK_MONOTONIC, &start);
    write(fd_null, buf, block_size);
    clock_gettime(CLOCK_MONOTONIC, &end);
    double latency = timespec_diff(start, end);
    fprintf(csv_write, "%.2f\n", latency);
}
```

Die Funktion *timespec_diff()* ist ebenfalls als *static inline* definiert.
Auch hier wird der Overhead für den Eintritt und den Exit des Syscalls (Kontextwechsel), der Overhead für die VFS (hier ext4) Layer und der Overhead für die /dev/null Treiberoperationen gemessen. Insbesondere fallen durch die Verwendung von /dev/null keine Hardwarelatenzen an.
Die Messungen haben die folgenden Ergebnisse geliefert:

/



Count: 1,000,000
Mean: 211.50 ns
Median: 210.00 ns
Std Deviation: 80.28 ns
Min: 200.00 ns
Max: 10621.00 ns
95% CI: [211.34, 211.66] ns
50th Percentile: 210.00 ns
95th Percentile: 230.00 ns
99th Percentile: 270.00 ns

/

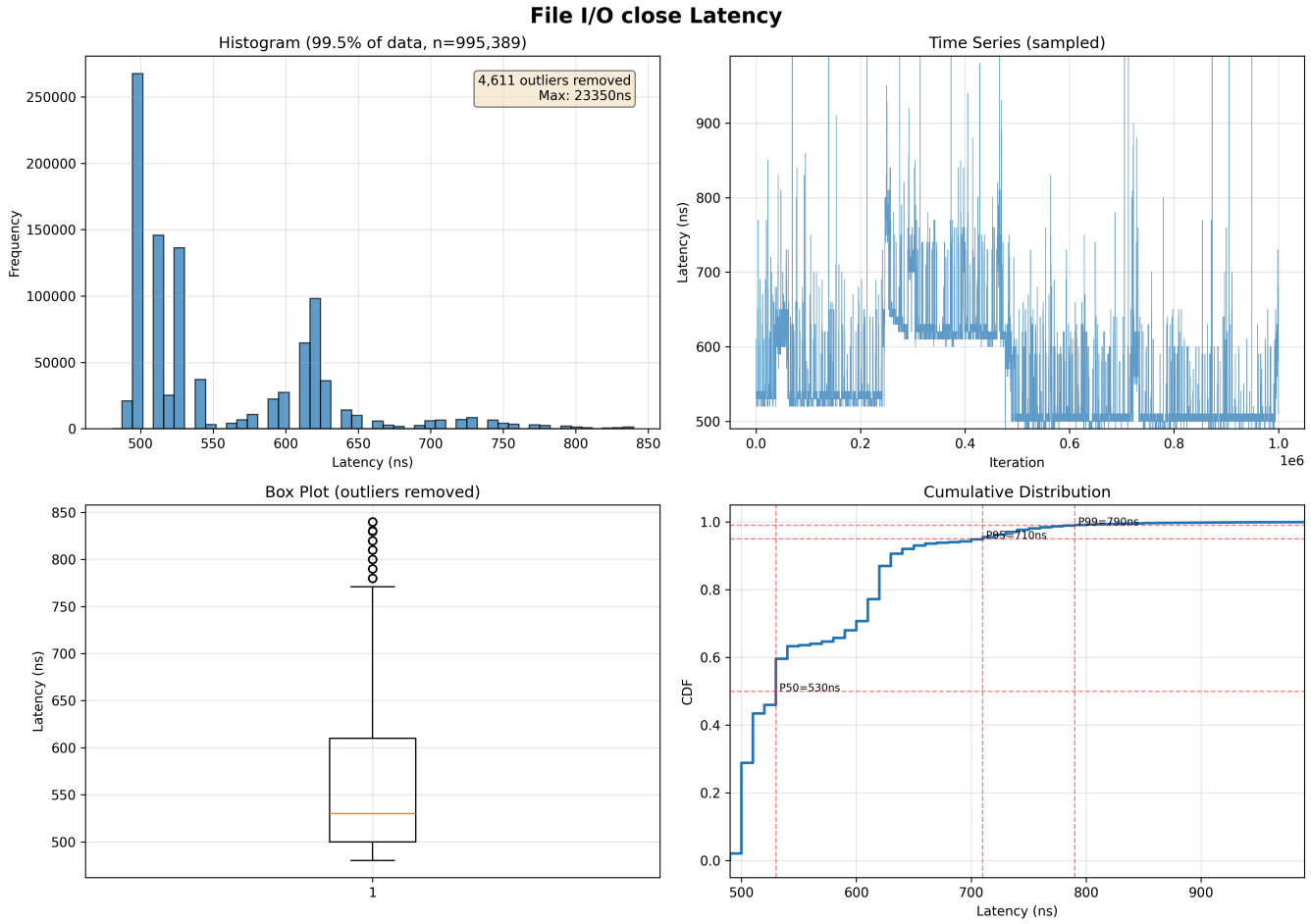
close

Die Latenz einer close-Operation wurde mit dem Program *filesystem.c* in 1.000.000 Durchläufen gemessen und in eine .csv Datei ausgegeben.
Auch für die Messung der Latenz der close-Operation wurde der Prozess an einen festen Kern gebunden und ein WarmUp durchgeführt (vgl. open).
Es wurde ebenfalls *clock_gettime()* für die einzelnen Zeitmessungen verwendet und das jeweilige Ergebnis anschließend in eine csv Datei geschrieben:

```
for(int i = 0; i < iters; i++){

// open block vgl. oben

...

clock_gettime(CLOCK_MONOTONIC, &start);
close(fd_null);
clock_gettime(CLOCK_MONOTONIC, &end);
latency = timespec_diff(start, end);
fprintf(csv_close, "%.2f\n", latency);
}
```

Die Funktion *timespec_diff()* ist ebenfalls als *static inline* definiert.
Auch hier wird der Overhead für den Eintritt und den Exit des Syscalls (Kontextwechsel), der Overhead für die VFS (hier ext4) Layer und der Overhead für die /dev/null Treiberoperationen gemessen. Insbesondere fallen durch die Verwendung von /dev/null keine Hardwarelatenzen an.
Die Messungen haben die folgenden Ergebnisse geliefert:

/



Count: 1,000,000
Mean: 560.68 ns
Median: 530.00 ns
Std Deviation: 157.35 ns
Min: 480.00 ns
Max: 23350.00 ns
95% CI: [560.37, 560.98] ns
50th Percentile: 530.00 ns
95th Percentile: 710.00 ns
99th Percentile: 790.00 ns

/

Spinlocks

Das Spinlock wurde über ein Kernel-Modul in der Datei *spinlock_kernel.c* realisiert. Die Latenzen wurden in 1.000.000 Durchläufen gemessen und in eine .csv Datei ausgegeben.
Es wurde die Latenz für das Erhalten und Freigeben eines uncontended Spinlocks - um den kleinstmöglichen Overhead im Kern zu messen - mithilfe der Funktion *ktime_get_ns()* in Nanosekunden gemessen:

static void measure_uncontended(void)

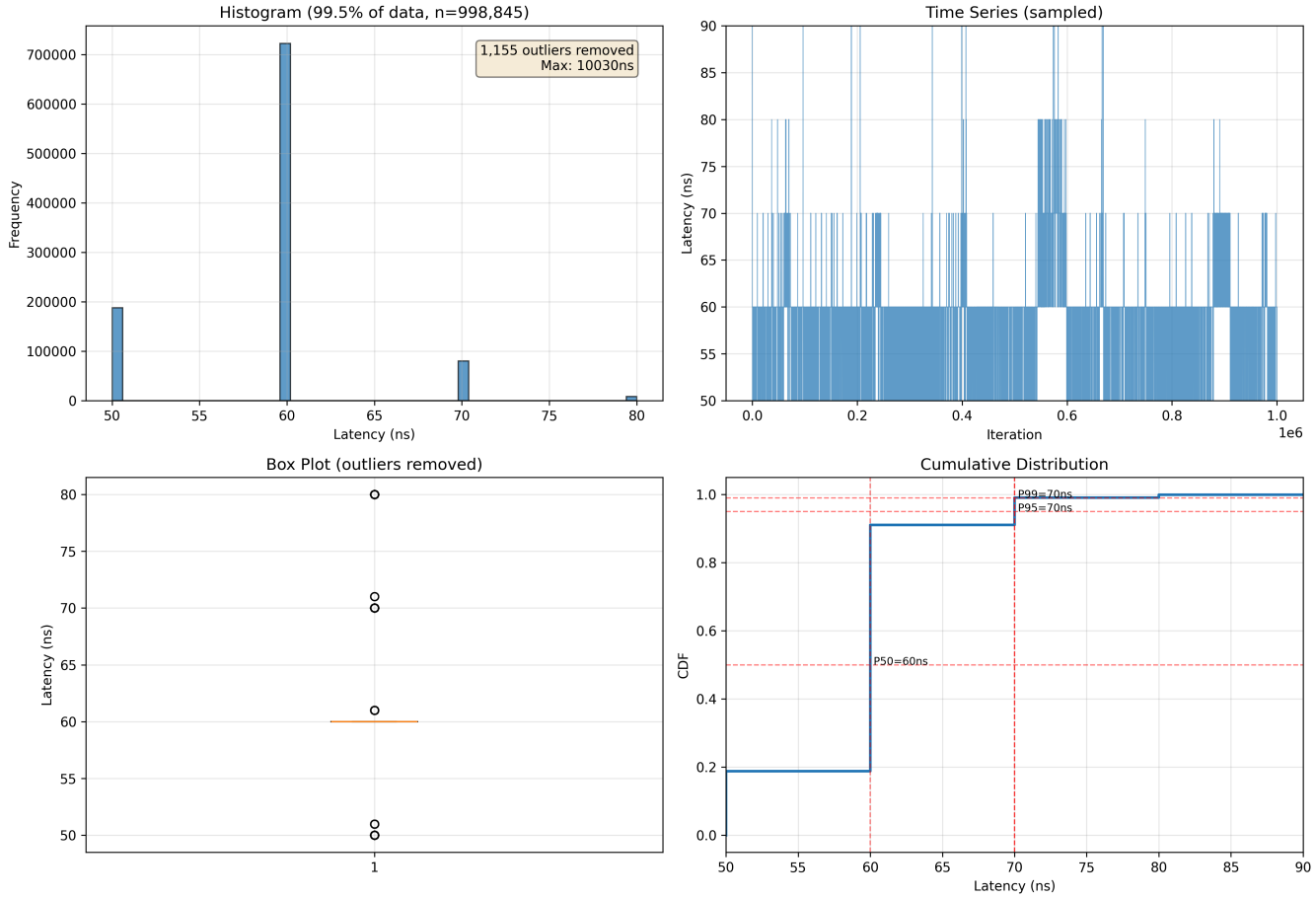
```
1       int i;
2       u64 start, end;
3       num_measurements = 0;
4
5       for(i = 0; i < ITERS; i++){
6           start = ktime_get_ns();
7           spin_lock(&lock);
8           spin_unlock(&lock);
9           end = ktime_get_ns();
10
11          measurements[i] = end - start;
12          num_measurements++;
13      }
```

Die Ausgabe der Messungen erfolgt mithilfe von *seq_printf()* an:

```
1   static int spinlock_proc_output(struct seq_file *m, void *v){
2       int i;
3       measure_uncontended();
4       for(i = 0; i < num_measurements; i++){
5           seq_printf(m,"%llu\n", measurements[i]);
6       }
7       return 0;
8   }
```

Die Initalisierung des Moduls erfolgt in der Funktion *static int __init spinlock_kernel_init(void).* Die Messung lieferte die folgenden Ergebnisse:

/



Count: 1,000,000
Mean: 59.53 ns
Median: 60.00 ns
Std Deviation: 47.32 ns
Min: 50.00 ns
Max: 10030.00 ns
95% CI: [59.44, 59.62] ns
50th Percentile: 60.00 ns
95th Percentile: 70.00 ns
99th Percentile: 70.00 ns

/

Semaphore

Die Latenzen für das blockierende Warten wurden mithilfe der der Datei *semaphore.c* gemessen. Das Program misst die Latenz für das Aufwecken einer Semaphore in 1.000.000 Durchläufen mithilfe von *clock_gettime()*:

```
for(int i = 0; i < ctx->count; i++){
    // measurement
    clock_gettime(CLOCK_MONOTONIC, &start);
    sem_wait(ctx->req);
    sem_post(ctx->ack);
    clock_gettime(CLOCK_MONOTONIC, &end);
    ctx->measurements[i] = timespec_diff(start, end);
}
```

Allerdings wartet der weckende Thread für $50\mu s$ bevor er den messenden Thread aufweckt, was die gemessene Zeit um $50000ns$ verfälscht:

```
for(int i = 0; i < iters; i++){
    usleep(50);
    sem_post(&req);
    sem_wait(&ack);
}
```

Um Störungen auszuschließen, die durch die Verwendung unterschiedlicher Kerne enstehen, sind der messende Thread und der weckende Thread an jeweils verschiedene Kerne gebunden mithilfe von:

```
static void pin_to_cpu(int cpu){
    cpu_set_t set;
    CPU_ZERO(&set);
    CPU_SET(cpu, &set);
    if(sched_setaffinity(0, sizeof(set), &set) != 0)
        perror("sched_setaffinity");
}
```

Außerdem wird ein Warmup durchgeführt, um Störungen bei der ersten Messung zu eliminieren, die z.B. durch das Caching von Instruktionen und Datenstrukturen im Kern entstehen:

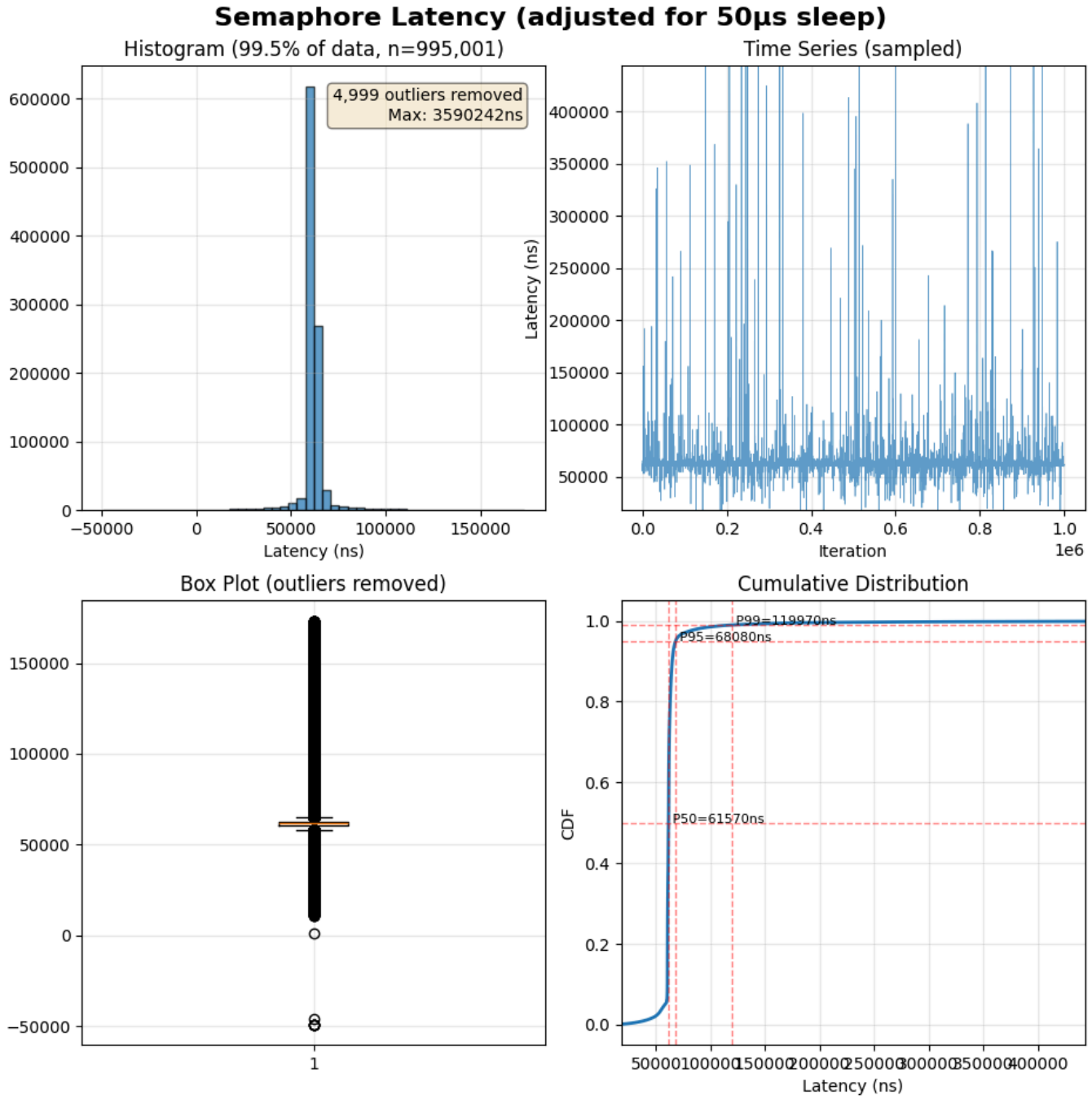
weckender Thread

```
// Warmup
for(int i = 0; i < 100; i++){
    usleep(50);
    sem_post(&req);
    sem_wait(&ack);
}
```

mesender Thread

```
// Warmup
for(int i = 0; i < 100; i++){
    sem_wait(ctx->req);
    sem_post(ctx->ack);
}
```

Das Program misst den Overhead der Syscalls für *sem_wait* und *sem_post*, den Overhead fpr den Kontextwechsel, die Latenz für die Kommunikation zwischen CPU 0 und CPU 1 und den Overhead, der durch die $50.000ns$ Wartezeit des weckenden Threads entsteht. Die Ergebnisse sind:



Count: 1,000,000
Mean: 63818.73 ns
Median: 61570.00 ns
Std Deviation: 30468.09 ns
Min: -49230.00 ns
Max: 3590242.00 ns
95% CI: [63759.01, 63878.45] ns
50th Percentile: 61570.00 ns
95th Percentile: 68080.00 ns
99th Percentile: 119970.01 ns

/

Pipe

Die Latenz für die *write()* und *read()* Operationen in / aus eine/r Pipe wird in *pipe.c* in jeweils 1.000.000 Durchläufen gemessen. Das Program implementiert eine nicht blockierende Pipe und ist *single threaded*. Dadurch wird der Overhead durch blockierende Syscalls, der Overhead durch Kontextwechsel zwischen verschiedenen Threads und der Overhead durch den Scheduler vermieden.

```
fcntl(pipefd[0], F_SETFL, O_NONBLOCK);
fcntl(pipefd[1], F_SETFL, O_NONBLOCK);
```

Um Störungen zu minimieren, wird der Prozess - wie in den übrigen Programmen - an einen festen Kern gebunden:

```
static void pin_to_cpu (int cpu){
    cpu_set_t set;
    CPU_ZERO(&set);
    CPU_SET(cpu, &set);
    if(sched_setaffinity(0, sizeof(set), &set) != 0)
        perror("sched_setaffinity");
}
```

Außderm findet jeweils ein *write()* und ein *read()* als Warmup statt, damit die entsprechenden Instruktionen und Datenstrukturen bereits in die entsprechenden Caches und Puffer geladen sind, bevor die Zeitmessung stattfindet:

write()

```
//warmup
write(pipefd[1], buffer, message_length);
```

read()

```
// warmup
read(pipefd[0], buffer, message_length);
```

Die Zeitmessung selbst erfolgt erneut mithilfe von *clock_gettime()*

write

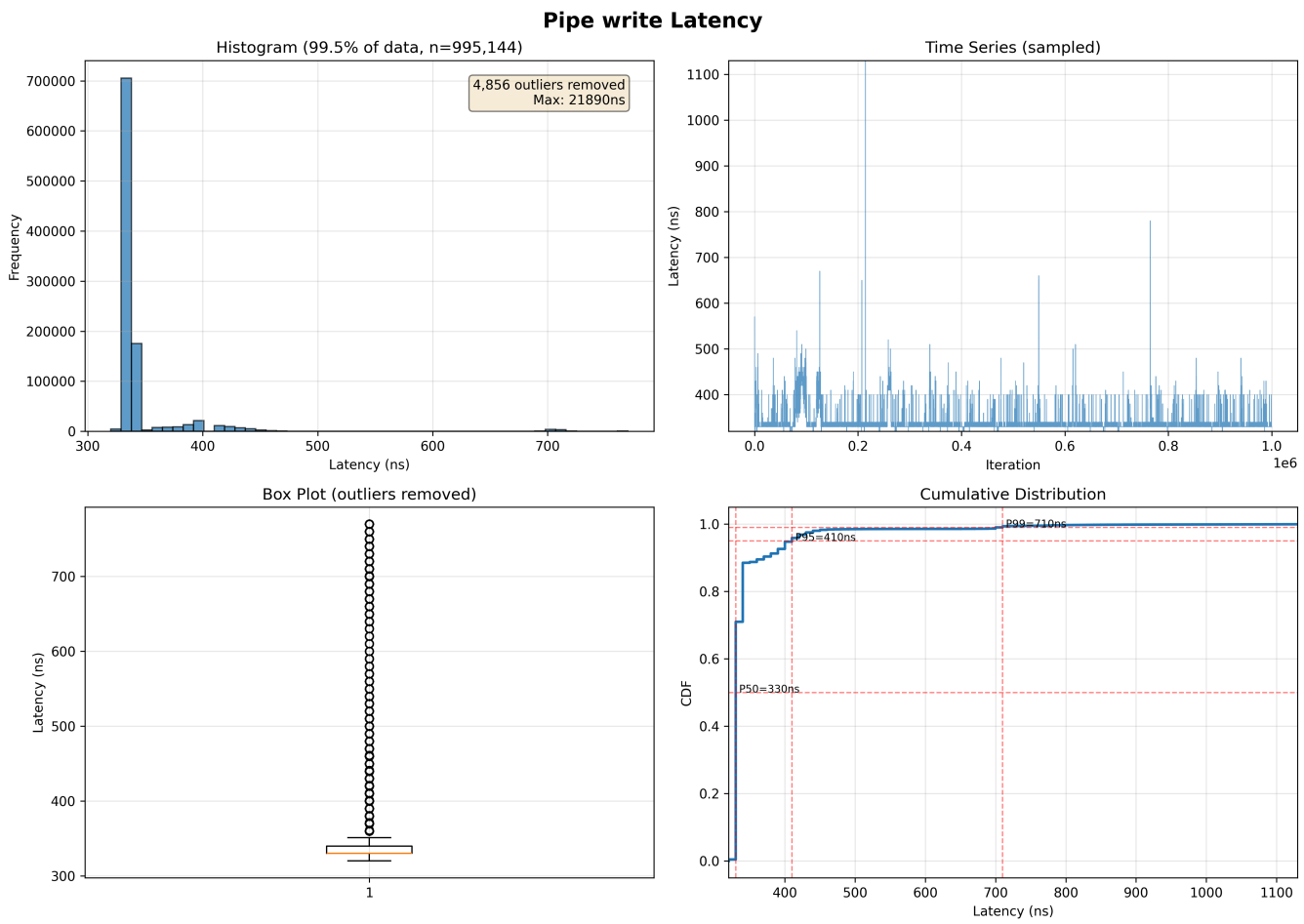
```
for(int i = 0; i < iters; i++){

    clock_gettime(CLOCK_MONOTONIC, &start);
    ssize_t n = write(pipefd[1], buffer, message_length);
    clock_gettime(CLOCK_MONOTONIC, &end);

    if (n < 0) {
        // drain pipe
        char drain[65536];
        read(pipefd[0], drain, sizeof(drain));
        i--;
        continue;
    }
    write_measurements[i] = timespec_diff(start, end);
}
```

read

```
for (int i = 0; i < iters; i++){

    write(pipefd[1], buffer, message_length);
    clock_gettime(CLOCK_MONOTONIC, &start);
    ssize_t n = read(pipefd[0], buffer, message_length);
    clock_gettime(CLOCK_MONOTONIC, &end);

    if(n != message_length){
        fprintf(stderr, "read error at iteration %d\n", i);
        break;
    }
    read_measurements[i] = timespec_diff(start, end);
}
```

Für den *write()* Syscall wird durch das Program der Syscall Overhead für die Kontextwechsel zwischen User- und Kernelspace, der Overhead, der durch den Pipe Puffer entsteht (z.B. Daten kopieren, Pointer updaten), der Overhead, der durch das Kopieren der Nachricht im Speicher entsteht, und der Overhead, der durch die VFS layer entsteht (Ext4 im Test), gemessen. Für den *read()* Syscall wird durch das Program der gleiche Overhead gemessen.
Grundsätzlich ist es möglich einzelne Teile dieses Overheads zu isolieren, z.B. kann der Overhead, der durch den Zustand des Pipe Puffers entsteht, isoliert werden, indem die Latenzen der Operationen für eine volle und für eine leere Pipe gemessen und verglichen werden. Analog kann der Overhead für das Kopieren einer Nachricht im Speicher isoliert werden, indem die Latenz für verschiedene Nachrichtengrößen gemessen wird. Die Ergebnisse sind:

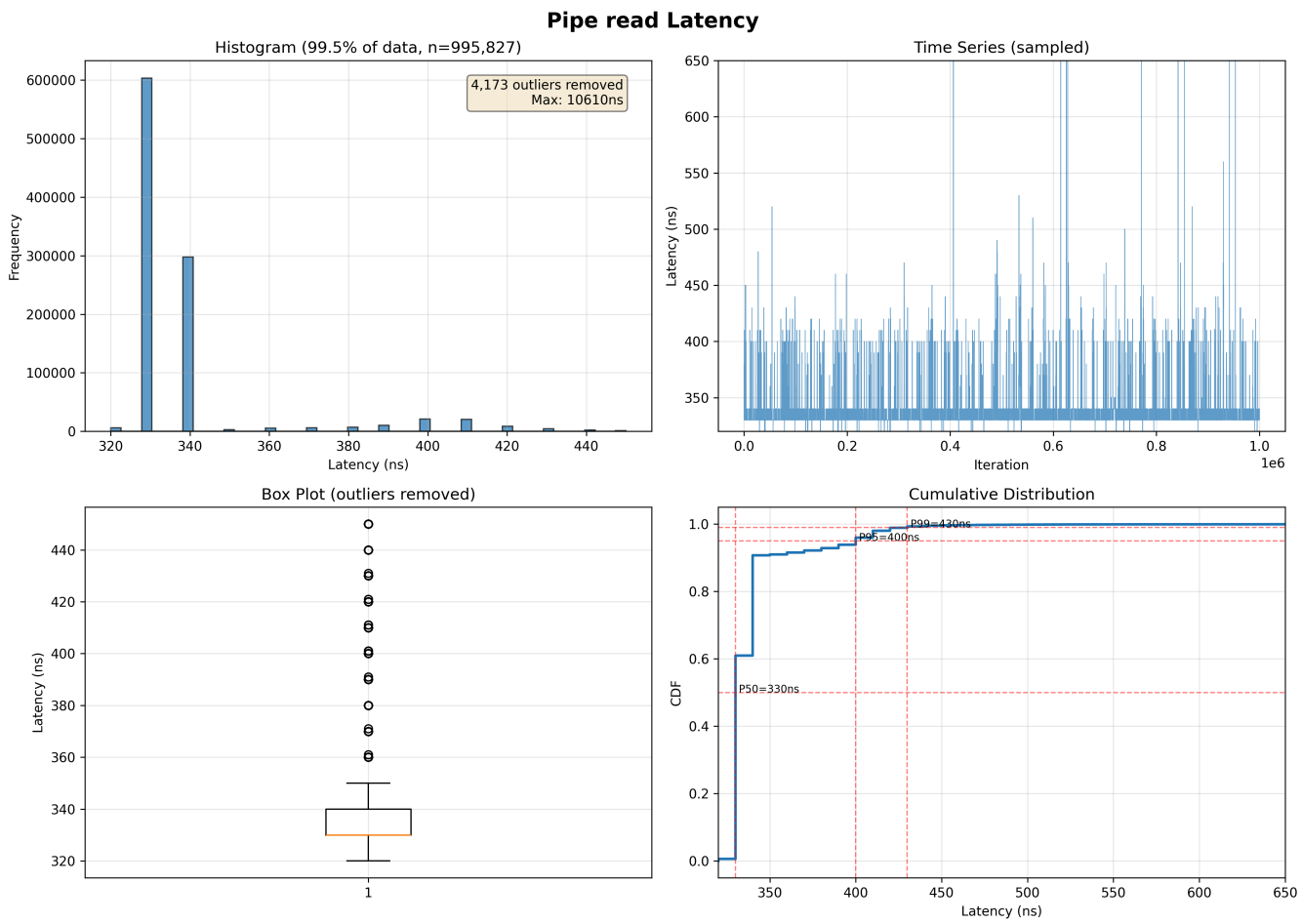
/



write

Count: 1,000,000
Mean: 347.13 ns
Median: 330.00 ns
Std Deviation: 121.30 ns
Min: 320.00 ns
Max: 21890.00 ns
95% CI: [346.89, 347.37] ns
50th Percentile: 330.00 ns
95th Percentile: 410.00 ns
99th Percentile: 710.00 ns

/



read

Count: 1,000,000
Mean: 341.61 ns
Median: 330.00 ns
Std Deviation: 102.54 ns
Min: 320.00 ns
Max: 10610.00 ns
95% CI: [341.41, 341.81] ns
50th Percentile: 330.00 ns
95th Percentile: 400.00 ns
99th Percentile: 430.00 ns

/

Vergleich

Die Latenz ist für das *Spinlock* am geringsten, da dieses einerseits nicht blockiert und es andererseits als Kernel Modul implementiert ist, sodass kein Overhead durch Kontextwechsel vom Userspace in den Kernelspace und zurück anfallen.

Etwas höher ist die Latenz für die *OWRC Operationen*, da u.a. durch die Kontextwechsel von Userspace in Kernelspace und zurück mehr Overhead anfällt. Allerdings fällt durch die Verwendung von */dev/null* kein Hardware bedingter Overhead an. Sodaß die Messungen verhältnismäßig nah an die Latenzen für die entsprechenden Syscalls herankommen.

Dagegen sind die gemessenen Latenzen für die die *Semaphore* am höchsten, da hier einerseits blockierend gewartet wird (für $50\mu s$) und andererseits ein erheblicher Overhead anfällt, der sich u.a. aus dem Overhead aus der Verwendung mehrerer Threads sowie dem Overhead durch den Scheduler und dem Overhead durch die Kontextwechsel von Userspace in Kernelspace und zurück ergibt. Das heißt, die Messungen sind durch den mitgemessenen Overhead viel höher als die tatsächlichen Kosten des Syscalls.

Geringer aber dennoch ebenfalls zu hoch sind die gemessenen Latenzen für die Pipe Operationen, die wie bereits erwähnt ebenfalls einen erheblichen Overhead mit messen, wie der Overhead für die Kontextwechsel zwischen User- und Kernelspace, der Overhead, der durch den Pipe Puffer entsteht , der Overhead, der durch das Kopieren der Nachricht im Speicher entsteht, und der Overhead, der durch die VFS layer entsteht.

Um die Messungen insbesondere für die Pipe Operationen und die Semaphore zu verbessern und die verschiedenen Overheads zu isolieren, müssten Vergleichsmessungen durchgeführt werden mit z.B. verschiedenen Nachrichtengrößen.