# ESCA: Effective System Call Aggregation for Event-Driven Servers

Yu-Cheng Cheng[†], Ching-Chun (Jim) Huang[*], Chia-Heng Tu[‡]
[*†‡]Department of Computer Science and Information Engineering, National Cheng Kung University
[*†‡]Tainan City, Taiwan (R.O.C.)
[*]jserv@ccns.ncku.edu.tw, [†]yucheng871011@gmail.com, [‡]chiaheng@ncku.edu.tw

***Abstract*—The switches between a non-privileged application and the OS kernel running in the CPU's supervisor mode have been inducing performance costs despite manufacturer efforts to provide special instructions for such transition. Software that heavily interacts with the underlying OS (e.g., I/O intensive and event-driven applications) suffers from system call overhead. To deteriorate this situation, security vulnerabilities in modern processors have prompted kernel mitigations that further increase the transition overhead. Particularly system-call–heavy applications have been reported to be slowed down by up to 30% with kernel page-table isolation (KPTI), the widely deployed mitigation for the Meltdown vulnerability. To decouple system calls from mode transitions, we revisit an old idea known as system-call batching or multi-calls: the bundling of system calls into a combined call, which only incurs the mode-transition costs of a single one. And then, we have implemented ESCA scheme to adapt system-call batching to Linux-based servers in the light of Meltdown and Spectre, effectively eliminating the slowdown of KPTI-affected applications. Our evaluation shows that the throughputs of real-world applications, benefiting from ESCA, can be improved with only 2 lines of code changed respectively: Nginx by up to 12%, lighttpd by up to 23%, and Redis by 4%. Meanwhile, using aggregated transitions, our approach allows faster system calls interleaved with full compatibility but without requiring Linux kernel patches.**

## I. Introduction

An event-driven architecture refers to a system composed of loosely coupled microservices. The creator (source) of the event only knows the event transpired but has no knowledge of the event's subsequent processing or the interested parties [1]. In modern HTTP servers, I/O event notification system calls such as `epoll` on Linux and `kqueue` on FreeBSD are used for listening to whether there are incoming events, which will be designated to the corresponding event handlers. Event-driven-based applications are quite sophisticated to be modeled: concurrent and keep-alive connections requests to the web server result in a mix of CPU-bound and I/O-bound processes. In addition, the event-driven architecture is composed of multiple microservices. The state-transition of them is largely unpredictable, which implies a grand challenge to optimize or instrument such non-deterministic applications [2]. To take Nginx as an example, it is one of the leading web servers, capable of representing highly concurrent event-driven applications. Profile-guided optimization (PGO) shipped with the state-of-the-art optimizing compilers is known to improve most software packages except Nginx – Yuan et al. reported that the degradation rates were about 0.59% [3]. It means that the execution of event-driven applications may not have a fixed execution path; hence the benefits brought from cache locality and branch prediction would be restrained.

A long-running event-driven application tends to face the challenge of a sudden burst of requests and gets involved in the system call (aka syscall), especially for I/O operations. Context switches between an unprivileged user process and the OS kernel running in the CPU's supervisor mode have typically been costly. Event-driven servers, classified as syscall-heavy applications, suffer from performance problems, caused by massive mode switches [4], [5]. KPTI dramatically increased this cost: directly, by switching the active page table at each context transition (i.e., two page-table switches per syscall), and indirectly by the subsequent, longer-lasting performance penalty from the TLB flushes triggered by these address space changes [6]. In the worst case, it causes 12% overhead to Redis, 15% to Apache, 12% to Nginx, and 2% to MongoDB [7]. In consequence, particularly syscall–heavy applications have been reported to be slowed down to 30% on KPTI-protected systems [8].

From the perspective of an application, each CPU cycle spent on the syscall interface is wasted and cannot be used for application-specific tasks. An old idea, "syscall batching" known as "multi-calls," was introduced to reduce syscall overhead: the bundling of several successive but potentially not directly related syscalls into a combined one, which only incurs the kernel-transition costs of a single syscall [9]–[12]. Nevertheless, plain syscall batching has some disadvantages like an increased total completion time that make this technique less attractive for event-driven applications. The main objective of this work is to reduce the per-syscall overhead through an advanced form of syscall aggregation, which is more flexible and applicable to service-oriented scenarios, offering full compatibility with the existing OS kernels and syscall interfaces. The contributions of this paper are:

- We propose and implement an effective syscall batching scheme to decrease the number of kernel boundary crossings, suitable for event-driven servers.
- Full syscall compatibility has been guaranteed by design in conjunction with real-world applications, including widely adopted web servers and key-value databases.
- An asynchronous and parallel execution of syscalls is being evaluated to eliminate unwanted latency, not limited to within the same batch.

## II. Background And Related Work

### A. Asynchronous I/O v.s. Non-blocking I/O

From the perspective of Linux, the file descriptor can become non-blocking access by configuring its attribute to `O_NONBLOCK`. Non-blocking I/O only guarantees that when the data is not available, the procedure call will return immediately with `EAGAIN` (resource temporarily unavailable) as an error indication without additional kernel involvement. However, asynchronous I/O will not only return from the kernel immediately but also trigger the kernel thread to execute tasks in the background.

### B. Development of Linux Asynchronous I/O (AIO)

In data-intensive applications, synchronous I/O operations often become a performance bottleneck. To avoid time-consuming blocking operations, since Linux v2.6, native AIO was introduced as a kernel-level implementation, supporting direct I/O mode only. By means of buffered I/O, the io_uring in Linux v5.1 eliminates the most challenging limitations of previous work [13]. io_uring provides a pair of rings (submission queue and completion queue) as an effective communication channel between the user applications and kernel. Several core specialized threads are spawned and asynchronously execute the tasks in the submission queue. To handle data that is not ready in buffered, on the other hand, a new workqueue subsystem (`io-wq`) that queues work items. Instead of using existing VFS interface, io_uring abstracts a series of I/O operations that is more friendly to an asynchronous model. The io_uring is still growing with doubtful maturities, such as limited I/O operations and API stability for major application adoptions. Performance drops and the regression of io_uring has been spotted from Linux v5.7.15 to v5.7.16 and later tweaked in v5.11 [14].

### C. Compound System Calls

The frequent data movement across the user-kernel boundary often becomes the performance bottleneck of web servers. Compound System Calls (Cosy) [15] packs the data-intense section into the kernel and executes it with the zero-copy technique, bringng 20-80% performance improvements to non-I/O bound applications. However, since Linux v2.2, `read-write` can be replaced by `sendfile` syscall, having almost no data copy. It means that Cosy no longer makes sense to allocate shared buffers (which might cause overhead) to achieve zero-copy. Furthermore, Cosy-GCC only supports a subset of the C syntax, and the function call is not supported, which substantially negates the practicality. Meanwhile, Cosy adds several new fields to the kernel internal structure `task_struct`, which affects non-Cosy applications.

### D. Batching

Rather than reducing the execution time of a single syscall, both System Call Clustering (SCC) [16] and Jadhav et al. [10] try to shorten the execution time of multiple syscalls by batching. SCC searches the clusterable region and transforms several syscalls into a single syscall, thereby reducing the number of user-kernel boundary crossings and leading to overall performance improvement. Compiler facilities such as function inlining and loop unrolling are provided to extend the clusterable region. However, the size of allocated memory for clusterable regions is not constant. It is proportional to the total number of syscalls' parameters. Also, the composition of the clusters cannot contain non-syscall statements.

Jadhav et al. propose new syscalls `recvmmmsg()` and `sendmmmsg()` to allow for receiving and sending multiple messages from multiple sockets in a single syscall invocation. It is targeted specifically at applications with high concurrent socket connections and using the `sendmmsg`-like I/O function, and syscalls will be invoked only when the number of syscalls crosses threshold, which implies the starting time of syscall is unbounded. The method is only suitable for applications which are not sensitive to latency.

### E. FlexSC

Traditional syscalls are likely implemented as the synchronous model and will raise a software interrupt when they are invoked, causing the cache damange. FlexSC [17] decouples the execution of syscalls, by means of deferring and batching the execution of syscalls, to improve temporal locality. To reduce blocking time and consider load balance, FlexSC offloads the execution of syscalls by maintaining kernel threads that are bound on different cores. It conflicts with the Unix-style programming model, so a threading package is created to make syscalls work transparently.

Meanwhile, heavy modifications against operating system kernel and C runtime library are needed, involving about 1400 lines changed in Linux kernel and glibc. The execution behavior of all syscalls is turned into being asynchronous regardless of their blocking and synchronous essence, leading to unguaranteed compatibility concerns. About 300 lines were changed for Nginx and Memcached [18], preventing FlexSC from being widely adopted by Linux kernel community due to its highly specialized and intrusive nature.

### F. FlexSC v.s io_uring

The design concepts of FlexSC and io_uring are very similar: 1) the original blocking I/O are substituted for I/O-intensive applications; 2) the I/O operations will be designated to a specific kernel thread, and the I/O operations of the user space will return immediately, meaning non-blocking. In addition, core specialization is also a common strategy of the two. Threads have their affinity, which not only improves locality but also ensures scalability. Last, to prevent the kernel thread from occupying too much CPU cycle, threads of both will sleep if idle time is greater than a specific threshold. However, FlexSC uses a shared table as a communication channel between kernel space and user space, while io_uring uses submit queue and completion queue. I/O offloading brings benefits to I/O-intensive applications, whether in the aspect of CPU usage, data locality, or the number of exceptions. Nevertheless, it is not a trivial task to rewrite synchronous applications into asynchronous counterpart.

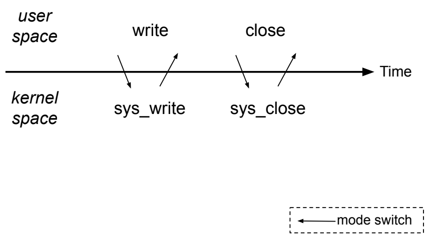

Figure 1. Each typical syscall undergoes two times of mode switch.

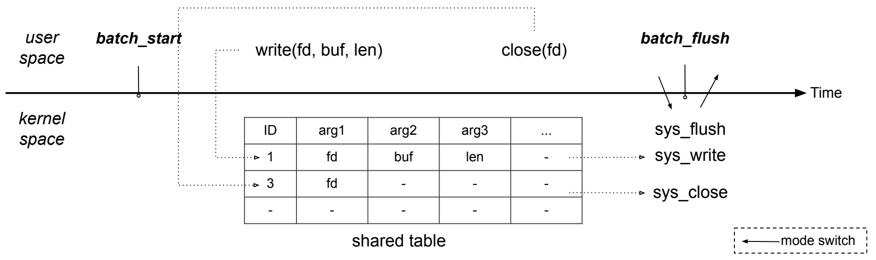

Figure 2. ESCA avoids syscalls raise exceptions in batching segment enclosed by batch_start and batch_flush. By information on shared table, it executes all of them in one syscall.

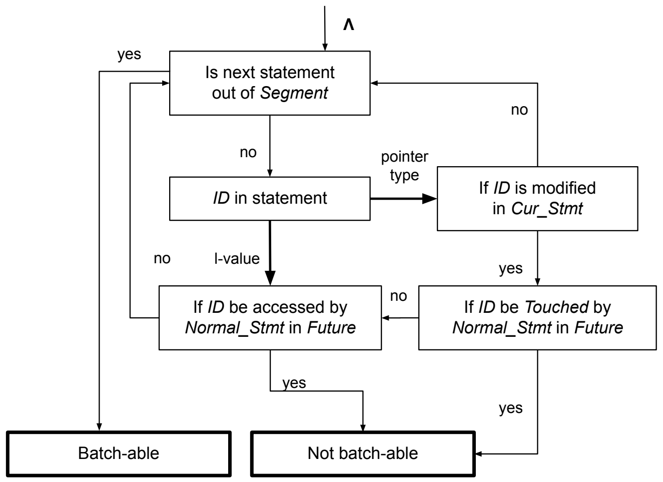
yes
Is next statement out of Segment
no
no
pointer type
ID in statement
If ID is modified in Cur_Stmt
no
l-value
yes
If ID be accessed by Normal_Stmt in Future
no
If ID be Touched by Normal_Stmt in Future
yes
yes
Batch-able
Not batch-able

Figure 3. Flowchart of determining if segment is batch-able

## III. Design

By decoupling syscalls and effectively reducing mode switch times, *Effective System Call Aggregation* (ESCA), the solution we proposed, can offer greater performance and flexibility than the traditional syscall interface. As shown in Figure 1, the number of mode switches is proportional to the number of syscalls. Considering event-driven applications, frequent mode switches hamper the performance. In our design (as shown in Figure 2), we provide two API calls, `batch_start` and `batch_flush`. The code section enclosed by them is called **batching segment**. It can appear more than one time in a single application. Compared with typical syscalls, ESCA eliminates mode switches in batching segments by decoupling syscalls. Instead of switching to the kernel or executing the corresponding service routine, syscalls in batching segment only record their syscall ID and arguments in the shared table. After `batch_flush` is called, ESCA finally switches to kernel mode, executes all syscalls in the shared table, and then switches back to user mode. No matter how many syscalls are there in batching segment, they are always handled by a single syscall invocation with two times mode switch. Hence, we'll explain the detailed ESCA working mechanism (Figure 4) in the following section.

### A. Valid Batching Segment

Due to putting off invocation of syscalls until flushing them, some statements may touch undetermined variables. The programmers are responsible for validating batching segments. To avoid this kind of dependency issue, we will discuss how to determine if the batching segment is batch-able. For the convenience of explanation, we define the following symbols:

- *Segment*: statements set enclosed by `batch_start` and `batch_flush`
- *Future*: code after current statement and before `batch_flush`
- *ID*: collection of syscall's arguments and return value
- *Cur_Stmt*: current statement
- *Normal_Stmt*: statement without syscall
- *Sys_Stmt*: statement with syscall
- *Touched*: be read or be modified

To examine if the *Segment* can be batched, we go through the flowchart shown in Figure 3. If any of the following conditions exists in *Segment*, then it is not batch-able:

1) Return value of syscall will be accessed in *Future*. Relative example is shown in Listing 1.
2) Pointer type argument is modified in current syscall and will be accessed by *Normal_Stmt* in *Future*. Relative example is shown in Listing 2.
3) Pointer type argument in syscall will be modified by *Normal_Stmt* in *Future*. Relative example is shown in Listing 3.

```
/* enter batching segment */
batch_start();
ret = close(fd);
/* return value will be accessed */
if (ret < 0) error_handling();
batch_flush();
/* start to execute all batched syscall */
```

Listing 1. Access undetermined l-value

```
/* enter batching segment */
batch_start();
/* buf will be modified */
read(fd, buf, 100);
/* buf will be accessed */
res = strcmp(buf, "foo");
batch_flush();
/* start to execute all batched syscall */
```

Listing 2. Access undetermined arguments

```
/* enter batching segment */
batch_start();
write(fd, buf, 100);
// buf will be modified by the original syscall
strcpy(buf, "foo");
batch_flush();
/* start to execute all batched syscall */
```

Listing 3. Override syscall's pointer type arguments

## B. *System Call Decoupling*

After calling syscall, the software trap will be triggered, then switch to kernel mode, looking for the corresponding trap service routine on the syscall table. It will switch back to user mode after finishing the service. To batch the syscalls, we must change the behavior of the original ones. First, we need to prevent syscalls in the batching segment from switching to kernel space. Second, we need to implement a new syscall that can execute cumulative syscalls at once.

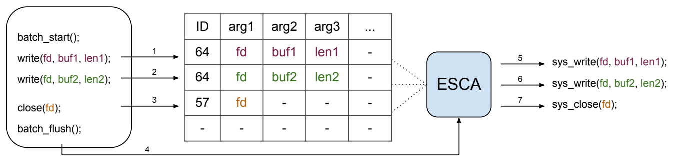

Figure 4. ESCA working mechanism

## C. *Shared Table Between Kernel and User Space*

To decouple syscall and got inspired by FlexSC, we need to create a mechanism to allow user space and kernel space to communicate with each other. We design the **shared table** to take care of this and it is composed of 64 entries. If the number of batched syscalls is greater than 64, they will be flushed internally. Each entry is 64 bytes and consists of several fields, including syscall ID, the array with six elements, and return values of syscall. The reason we design the size of an entry as 64 bytes is that it makes the size of the shared table exactly be one page; hence the data we access will scarcely be evicted from the TLB.

Take Figure 4 as an example. After entering batching the segment, the behavior of syscalls will be changed. Syscall `write` and `close` neither switch to kernel mode nor invocate relative syscall handlers. Instead, they only record their syscall ID and arguments to the shared table. After leaving the batching segment, `batch_flush`, a user-level syscall wrapper, was triggered. It helps the procedure to trap into the kernel and execute the kernel function in ESCA, which traverses the shared table and serially invocates syscall handlers. In our example, the syscall handler for `write` can be found by accessing the 64th member in `sys_call_table` (the symbol represents syscall table in Linux).

## IV. Implementations

In this section, we will explain the reasons why we encapsulate kernel modifications into a kernel module, how we let kernel and user efficiently communicate with each other, and the adaptation for executing syscalls.[1]

## A. *Kernel Module*

To implement batching syscalls efficiently, kernel changes are needed alongside the changes against event-driven applications. For the sake of compatibility and potential security concerns, this work concentrates on the principle of least privilege for OS kernel changes to minimize the privileged threat surfaces. We pack kernel changes into a kernel module, which can prevent applications that do not use ESCA from being affected. Besides, compiling the kernel module takes a shorter time than that of Linux kernel which implies the kernel module helps improve development efficiency.

## B. *Zero Copy*

In Linux, we cannot access the user address space in kernel mode. Although we can achieve it by using a function like `copy_from_user`, it impacts the performance more seriously as the size of copied data increases. In our implementation, we use `get_user_pages` to bind the page to the physical memory, and use `kmap` to map the physical pages to the kernel address space. In this way, data sharing is without the copy and the procedure is a one-time allocation.

## C. *Syscall Wrapper*

To change the behavior of the syscall, when the application is executed, the syscall wrapper of glibc is replaced with our shared library through `LD_PRELOAD`. Instead of trapping into kernel space and doing the trap service routine, the syscall wrapper only records specific information now. Because this approach ingeniously removes the restrictions of the batching segment, interleaving non-syscall logics and full C programming compatibility are supported.

## D. *Syscall Hooking*

We cannot add syscalls without modifying the Linux kernel [19]. What we can do is to replace an entry of the syscall table with our customized handler. It is crucial to make sure that the replaced entry is unused; otherwise, the system may crash after hooking the syscalls. `kallsyms_lookup_name` is no longer exported after Linux v5.7 but we can still locate the address of the syscall table through the kernel symbol table. Proper offset is needed by obtained address if kernel address space layout randomization (KASLR) [20] is enabled to prevent exploitation of memory corruption vulnerabilities. Also, it is necessary to clear the write protection bit of the control register if modifying the syscall table is required.

[1]The source is public: https://github.com/eecheng87/ESCA.

Two syscalls, `batch_register` and `batch_flush`, are introduced: the former will only be called once, responsible for mapping shared table and initialization, while the latter will execute the syscalls currently accumulated on the shared table at one time. Listing 4 shows the corresponding code.

```
long sys_batch(struct pt_regs *regs) {
  while (batch_table[i].rstatus == BUSY) {
    batch_table[i].ret =
        indirect_call(
          syscall_table[batch_table[i].sysnum],
          batch_table[i].nargs,
          batch_table[i].args);
    batch_table[i].rstatus = EMPTY;
    i = (i == table_max) ? 1 : i + 1;
  }
}
```

Listing 4. Code snippet of flushing handler

### E. Execute System Call Indirectly

After the decoupling syscall, it cannot be triggered from the wrapper provided by glibc. Instead, ESCA will invocate the syscall handler from the kernel space according to the type and parameters of the syscall on the shared table. Also, the address of the syscall handler can be retrieved by adding the offset to the syscall table.

| | **timeout** | **write error** | **latency ($\mu$s)** | **RPS** |
|---|---|---|---|---|
| lighttpd | 0 | 0 | 762.88 | 33402 |
| lighttpd-Async-ESCA | 0 | 0 | 724.53 | 35897 |

Table I. Integrate async-ESCA into lighttpd

## V. Experiment

In this section, we will discuss different event-driven applications and explain how to apply to ESCA. Also, we will evaluate them by both micro- and macro-benchmarks, measuring the overhead of mode switch, loading, and throughput.

Although ESCA can work in an asynchronous manner, the common event-driven applications are designed as a synchronous model. Converting them into an asynchronous model is not trivial. Also, the impact on performance is unknown. After primitive testing, we did not find asynchronous-ESCA can bring benefits from event-driven applications designed with a synchronous model (The experiment result is shown in Table I). As a result, all of the following experiments are dedicated to synchronous-ESCA.

### A. Experimental Environment

The experiments presented in this section are run on an AWS t4g.micro instance powered by Arm-based AWS Graviton2 processors with the characteristics shown in Table II. The reason we choose AWS Graviton2 processor is that its instances deliver up to 40% better price performance for all workloads over comparable x86-based instances. The limited number of cores will not affect our experiments, because both targets with and without ESCA are not scalable. As shown in Table III, the throughput does not grow as core numbers grow.

| Component | Specification |
|---|---|
| Cores | 2 |
| Memory | 1 GiB |
| Architecture | aarch64 |
| L1 caches | 64 KiB |
| L2 cache/vCPU | 1 MiB |
| LLC | 32 MiB |
| Kernel version | 5.11.0 |

Table II. Characteristics of AWS t4g.micro instance

| Target \ Cores | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| lighttpd | 82,184 | 79,819 | 81,460 | 78,583 |
| lighttpd-ESCA | 99,115 | 95,651 | 96,284 | 96,261 |

Table III. Throughput with different core numbers

### B. Benchmarking

*1) Mode Switch Overhead:* The mode switch consists of entering to the kernel and returning from the kernel, and both of them are similar. For example, the overhead of entering kernel includes flushing user-mode pipeline, storing user-mode registers, restoring kernel-mode register, etc.

To evaluate the overhead, we create a new syscall whose handler will return immediately. The time elapsed of executing it is almost equal to the time of mode switch. In our experiment, a mode switch (enter and exit) takes 800 $\sim$ 1000 CPU cycles (with Meltdown mitigation). In other words, it takes 300 $\sim$ 370 ns under the machine with a 2.7 GHz clock.

*2) Throughput:* It is a very common way to evaluate the performance of event-driven applications by throughput. Higher throughput means that applications can handle more requests. For web server, we use a modern HTTP benchmarking tool `wrk` [21] to generate the HTTP payloads. We will get the average rate of the incoming requests (RPS) from it. For Redis, we use its built-in benchmarking tool `redis-benchmark` which also uses RPS as the basis for evaluation.

*3) Load:* Aside from throughput, the loading of applications is also an important basis for evaluation. The way we evaluate loading is to measure the time it takes in main loop to complete a certain number of requests. Also, the experiments will be done with different connection numbers (from 100 to 200). If the time in the loop is less, it means that the CPU can take its time elsewhere and improve the overall efficiency.

### C. Real-World Applications

Most of the web servers have the characteristics in common: (1) an infinite loop; (2) using `epoll`-like syscall to listen to the events on the ready list; (3) with corresponding event handlers to handle ready events.

In the following experiments, we run `wrk` with two threads because of our machine's core number. We choose 50 as connection numbers because of the design of ESCA. When the shared table is full, all batched syscalls on it will be flushed implicitly. This means that setting the connection number to a number over 64 will not make any difference. First, we measure the distribution of the throughput with small payloads which sizes are between 1 KiB and 40 KiB. Second, we

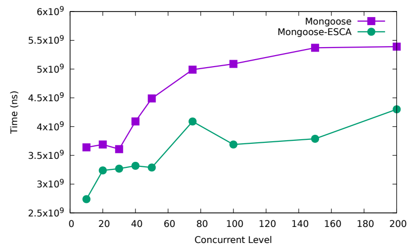

Figure 5. Load of Mongoose main loop

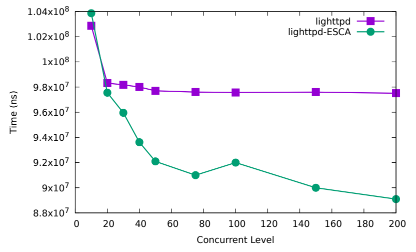

Figure 6. Load of lighttpd main loop

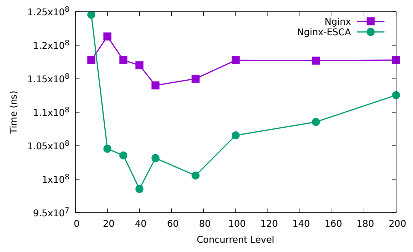

Figure 7. Load of Nginx main loop

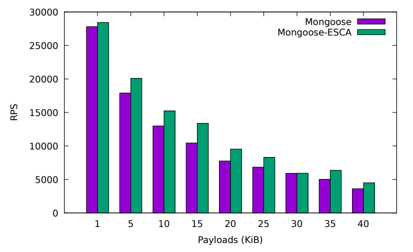

Figure 8. Throughput of Mongoose with small payloads

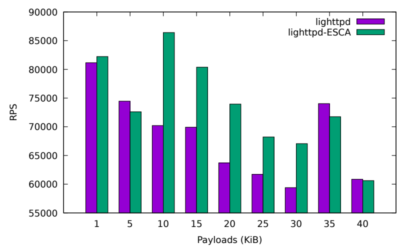

Figure 9. Throughput of lighttpd with small payloads

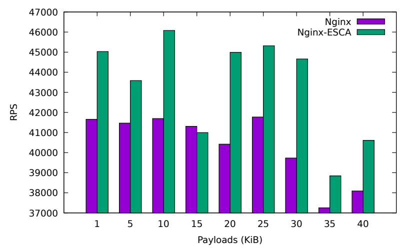

Figure 10. Throughput of Nginx with small payloads

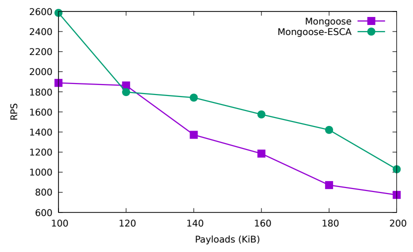

Figure 11. Throughput of Mongoose with large payloads

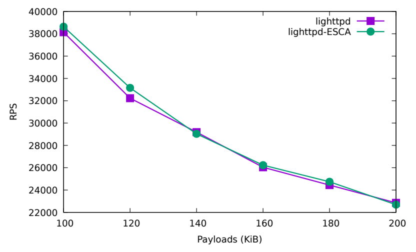

Figure 12. Throughput of lighttpd with large payloads

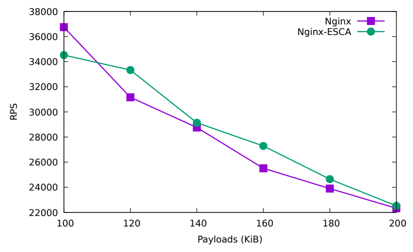

Figure 13. Throughput of Nginx with large payloads

measure them with large payloads which sizes are between 100 KiB and 200 KiB. The reason we design two types of experiments to measure the throughput of applications is that the performance of applications shows different trends in these two scenarios. Details will be covered in the following sections.

*1) Mongoose:* As a start, we selected Mongoose[2], which provides easy used and robust API for users to write a web server. In this experiment, we mark the for loop in `mg_mgr_poll` as the batching segment. As shown in Figure 8, the throughput of Mongoose-ESCA transmitting small payloads, in the best case and the average case, can be improved by 22% and 13% respectively. On the other hand, Figure 11 shows that transmitting large payloads with Mongoose-ESCA can increase the performance by 62% in the best case. Considering applications load, Figure 5 shows that Mongoose-ESCA takes less time in the main loop between connection numbers from 10 to 200. No matter from the perspective of throughput or loading in the main loop, ESCA can improve both of them. The results are under our expectations because ESCA effectively reduces the number of user-kernel crossings.

[2]CivetWeb was forked from *Mongoose* in 2013, before the later changed its licence from MIT to commercial + GPL. The mission of *CivetWeb* is to provide easy-to-use and powerful embeddable web server with optional CGI, SSL and Lua support. They both are widely used, and ESCA is beneficial to them as well.

*2) lighttpd:* The next target is lighttpd, a bigger single-threaded high-performance web server. Like most event-driven web servers, lighttpd listens to incoming events on epoll-instance and continues to do corresponding event handling. In this experiment, we use lighttpd v1.4 and mark the for loop in `main_server_loop` as the batching segment. Figure 9 shows that the throughput of the lighttpd-ESCA web server has improved at most 23% in small payloads cases (8% on average). The default network-backend-write operation of lighttpd was set to `writev`. To enhance the performance, it uses `sendfile` to handle write operations if payloads are greater than 35 KiB. Although we don't find lighttpd-ESCA has improvement in large payloads scenarios (as shown in Figure 12), we also don't find any regression. The reason why ESCA cannot bring benefits to large payloads is caused by **batching latency**. Under the batching strategy, the execution of the syscalls will be deferred and will only be executed after flushing. This side effect will be detailedly explained in the following section. Besides batching latency, ESCA itself has overhead (recording syscalls' arguments) to applications.

With the fewer connection numbers, the fewer syscalls can be batched and fewer benefits can be brought from ESCA. As shown in Figure 6, when the number of connections in lighttpd-ESCA is greater than 20, the benefits brought from ESCA start to be greater than its overhead.

*3) Nginx:* Nginx was designed with high-performance, lower footprint, multi-threading and event-driven concurrency in mind. The master process of Nginx is responsible for initializing and managing worker processes, while the worker process is responsible for processing new connections and transmitting request payloads. Nginx, on Linux, also uses `epoll` to listen to events. However, regardless of the size of the requested file, it is transmitted through `sendfile`. In this experiment, we use Nginx v1.20.1 and mark the for loop in `ngx_epoll_process_events` as the batching segment. Figure 10 shows that the throughput of Nginx-ESCA transmitting small payloads, in the best case and the average case, can be improved by 12% and 7% respectively. Similar to lighttpd, Nginx does not gain significant improvements in large payloads scenarios (as shown in Figure 13). Figure 7 shows that when the number of connections in Nginx-ESCA is greater than 20, the loading in the loop begins to get the upper hand.

*4) Key Value Database:* Besides web servers, key-value databases, data storage paradigm designed for storing and retrieving, are illustrated. Performance is an important concern for key-value databases since they have to handle client requests effectively. Redis, one of the well-known key-value databases, uses `epoll` to listen for events and processes events through `read` and `write`. In this experiment, we use Redis v6.2.5 and mark the for loop in `handleClientsWithPendingWrites` as the batching segment. It is evaluated by built-in Redis-benchmark, which can be used to simulate an arbitrary number of clients connecting at the same time and performing actions on the server, measuring how long it takes for the requests to be completed. We set the connections number as 100, the pipeline number as 16, and the total number of requests as 100000. The result of experiments shows that Redis-ESCA can improve 3% and 4% performance in `SET` and `GET` respectively.

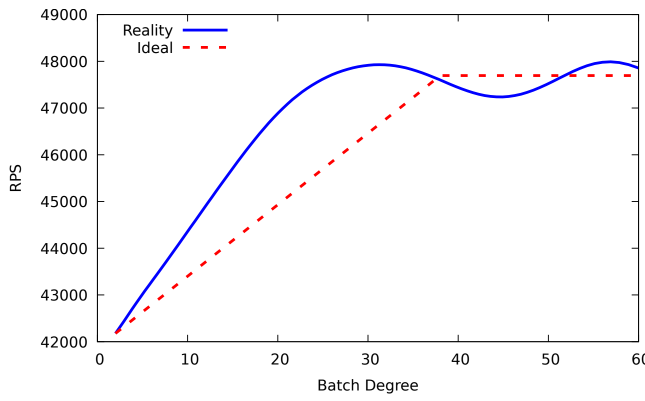

Figure 14. The ideal and real performance trend of Nginx-ESCA to different batching sizes

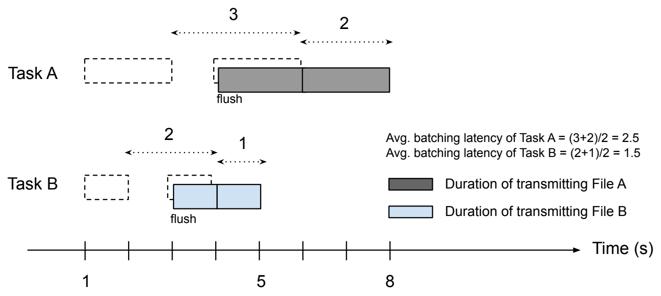
3
2
Task A
flush
2
1
Avg. batching latency of Task A = (3+2)/2 = 2.5
Avg. batching latency of Task B = (2+1)/2 = 1.5
Task B
Duration of transmitting File A
Duration of transmitting File B
flush
Time (s)
1
5
8

Figure 15. Batching latency brought from different payloads

## D. Modeling the Performance of ESCA-applications

In this section, we tried to model the throughput of ESCA-applications with different numbers of batching syscall. We define $D$ as batching degree which means the average numbers of syscall would be flushed each time, $C$ as convergent point, which equals to the average connections number times numbers of supported syscall type. If the result is greater than the size of shared table, $C$ should be set to the maximum available entry number of table. Further, we assume the throughput grows steadily before the convergent point and define $\alpha$ as a constant coefficient. Following is the equation which models ESCA-applications:

$$Throughput = \begin{cases} \alpha D, & \text{if } D < C \\ \alpha C, & \text{if } D \geq C \end{cases}$$

We take Nginx as a demonstration, given the connections number 20 (can be controlled by `wrk`), the average connection number is 19, the supported syscall types are 2 (`sendfile` and `close`), and the max table size is 64. Therefore, the convergent point should be 38 (average connections number times numbers of supported syscall type).

As shown in Figure 14, ideally, the performance of Nginx-ESCA grows steadily before the convergent point. After that, the performance will remain stable. In our experiment, we find that the real trend of the performance of Nginx-ESCA almost follows our model. Its performance stagnates near to ideal convergent point.

## VI. Limitations

## A. Manually Find Hot Spot

The user of ESCA is responsible for finding the syscall-intensive section in the applications and marking it as a batching segment. Besides, the user also needs to confirm whether the segment is valid: If yes, it is a batch-able segment. Due to the deferred execution of syscalls in ESCA, for correctness, we need to ensure that there are no dependency issues in the batching segment. Although it does limit the available scope, we still can alleviate the restrictions with some facts. We find that syscalls like `write`/`writev` scarcely fail, so we can assume the result is successful. Now, accessing undetermined l-value is suppressed and it really helps extend available scenarios.

## B. Large Payloads Transmission

It is very common to evaluate the server with throughput and latency. The former is critical to whether the manufacturer can simultaneously serve a large number of customers. The latter affects whether customers can receive a response as soon as possible. Severe latency implies that fewer customers can be served in the same period, which also means that throughput is reduced. Because of batching latency, we observe that ESCA would not improve the performance of transmitting large payloads as small payloads do. As shown in Figure 15, Task B handling smaller payloads suffers less overhead from batching latency. Although batching latency, in some scenarios, may lower the benefits brought from ESCA, compared to prior work with the provision of `recvmmmsg()` and `sendmmmsg()` [10], the batching latency of ESCA is bounded. Because accumulated syscalls will be executed by calling `batch_flush` at the latest, there are only the first few requests being delayed. It is a good deal to make: A trade-off between delays in few requests and overall performance (throughput and loading) improvement.

## VII. Conclusion

The main objective of this work was to reduce the per-syscall overhead through the use of effective syscall aggregation. For that purpose, ESCA takes advantages of system call batching and exploits the parallelism of event-driven applications by leveraging Linux I/O model to overcome the disadvantages of previous solutions. Although the current implementation needs further improvement, the evaluation showed that the main objective could be achieved: ESCA is capable of reducing the per-syscall overhead by up to 62% for embedded web servers. Real-world highly concurrent event-driven applications such as Nginx and Redis are known to benefit from ESCA, along with full compatibility with Linux syscall semantics and functionalities.

Our findings can be used by Linux developers for the sake of long-term syscall maintenance. In particular, with syscall aggregation mechanisms such as ESCA, the kernel APIs can be kept clean and elegant without the need of "macro" syscalls such as `sendmmsg` and `recvmmsg` which combine various kernel operations. If there was a batching syscall, the above could be implemented in user space, reducing the kernel complexity and still ensuring the performance.

### Future Work

To improve the throughput of applications when transmitting large payloads, multi-threaded offloading is an empirical method to achieve. This is also one of the challenges io_uring devoting to conquering. However, both the maturity of io_uring and the suitability of applying asynchronous I/O to applications with a synchronous model remain questionable. To maintain asynchronous I/O, it needs a blocking function such as `io_uring_wait_cqe` to wait for completion, which might ruin the design and hamper the performance of event-driven applications. Although our current design is compatible with asynchronous I/O, we are not confident if the latest highly concurrent event-driven applications based on synchronous I/O are fully capable of performing asynchronous I/O model transition without considerable engineering efforts. For future work, we will look for opportunities of consolidating efficient asynchronous I/O for service-oriented applications transparently.

## References

[1] B. M. Michelson, "Event-driven architecture overview," *Patricia Seybold Group*, vol. 2, no. 12, pp. 10–1571, 2006.

[2] C. S. Jensen, A. Møller, V. Raychev, D. Dimitrov, and M. Vechev, "Stateless model checking of event-driven applications," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 57–73, 2015.

[3] P. Yuan, Y. Guo, and X. Chen, "Experiences in profile-guided operating system kernel optimization," in *Proceedings of 5th Asia-Pacific Workshop on Systems*, 2014, pp. 1–6.

[4] T. Hruby, T. Crivat, H. Bos, and A. S. Tanenbaum, "On sockets and system calls: Minimizing context switches for the socket API," in *2014 Conference on Timely Results in Operating Systems (TRIOS 14)*. USENIX Association, 2014.

[5] C.-C. J. Huang and C.-F. Yang, "An empirical approach to minimize latency of real-time multiprocessor Linux kernel," in *International Computer Symposium (ICS)*, 2020, pp. 214–218.

[6] L. Müller, "KPTI a mitigation method against Meltdown," *Advanced Microkernel Operating Systems*, p. 41, 2018.

[7] Z. Hua, D. Du, Y. Xia, H. Chen, and B. Zang, "EPTI: Efficient defence against Meltdown attack for unpatched VMs," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 255–266.

[8] D. Hansen. (2017) KAISER: unmap most of the kernel from userspace page tables. [Online]. Available: https://lwn.net/Articles/738997/

[9] N. R. T. Horman, "Batch execution of system calls in an operating system," Patent US9 038 075B2, 2015.

[10] A. S. Rahul Jadhav, Zhen Cao, "Improved system call batching for network I/O," 2019.

[11] G. Kroah-Hartman. readfile: implement readfile syscall. [Online]. Available: https://lore.kernel.org/lkml/20200704140250.423345-1

[12] L. Gerhorst, B. Herzog, S. Reif, W. Schröder-Preikschat, and T. Hönig, "AnyCall: Fast and flexible system-call aggregation," in *Proceedings of the 11th Workshop on Programming Languages and Operating Systems*, 2021, pp. 1–8.

[13] J. Axboe, "Efficient IO with io_uring," 2019.

[14] J. Axboe. (2020) task_work: Use TIF_NOTIFY_SIGNAL if available. [Online]. Available: https://lore.kernel.org/lkml

[15] A. Purohit, J. Spadavecchia, C. Wright, and E. Zadok, "Improving application performance through system call composition," Citeseer, Tech. Rep., 2003.

[16] M. Rajagopalan, S. K. Debray, M. A. Hiltunen, and R. D. Schlichting, "System call clustering: A profile-directed optimization technique," *Technical Report*, 2002.

[17] L. Soares and M. Stumm, "FlexSC: Flexible system call scheduling with exception-less system calls," in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2010, pp. 33–46.

[18] L. Soares and M. Stumm, "Exception-less system calls for event-driven servers," in *USENIX Annual Technical Conference (ATC)*, 2011.

[19] C.-C. J. Huang, C.-H. Lin, and C.-K. Wu, "Performance evaluation of Xenomai 3," in *Proceedings of the 17th Real-Time Linux Workshop (RTLWS)*, Graz, Austria, 2015, pp. 21–22.

[20] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is dead: long live KASLR," in *International Symposium on Engineering Secure Software and Systems*, 2017, pp. 161–176.

[21] W. Glozer. (2018) wrk: a HTTP benchmarking tool. [Online]. Available: https://github.com/wg/wrk