CLC ____________ Number____________

UDC____________ Available for reference ☐Yes ☐No

Southern University of Science and Technology

Undergraduate Thesis

Thesis Title:	**Real-time capturing of system calls on ARM**
Student Name:	**Haonan Li**
Student ID:	**11712510**
Department:	**Department of Computer Science and Engineering**
Program:	**Computer Science and Technology**
Thesis Advisor:	**Fengwei Zhang**

Date: May 10, 2021

COMMITMENT OF HONESTY

1. I solemnly promise that the paper presented comes from my independent research work under my supervisor's supervision. All statistics and images are real and reliable.
2. Except for the annotated reference, the paper contents no other published work or achievement by person or group. All people making important contributions to the study of the paper have been indicated clearly in the paper.
3. I promise that I did not plagiarize other people's research achievement or forge related data in the process of designing topic and research content.
4. If there is violation of any intellectual property right, I will take legal responsibility myself.

Signature:

Date:

Real-time capturing of system calls on ARM

Haonan Li

(Department of Computer Science and Engineering   Advisor: Fengwei Zhang)

[**ABSTRACT**]: Bug diagnosis is difficult. The first step of bug diagnosis is to reproduce the bug. In areas such as application development, developers usually can only rely on the report logs uploaded by the user to try to reproduce bugs. Unfortunately, it is still challenging to reproduce bugs that occurred in the production environment at the development environment. The primary obstacle of reproduction is non-deterministic events at runtime, such as system calls. Hence, the same execution may lead to different results.

In this thesis, I present Syscord, a practical tool for recording system calls on Linux. Syscord utilizes Linux tracepoints to hook system calls. Syscord collects relevant information for each system call related to their effects, which further helps developers reproduce and fix bugs. I implement a prototype of Syscord and evaluate it with real-world applications. The result demonstrates the Syscord capturing system calls entirely and efficiently.

[**Keywords**]: Linux, Syscall, Record

Contents

1. Intoduction

The program often fails. To sufficiently understand and prevent failures, developers requires firstly reproduce these bugs, which ensures the same output and bugs. However, directly re-exection is not suitable for non-deterministic failures, as they may not appear in a re-execution procedure. Non-deterministic failures are the consequence of non-deterministic instructions.

Instructions for running a program can be divided into two categories. One is deterministic, which means the behavior of deterministic instruction is determined in each execution. The other type is non-deterministic, meaning that execution in different situations will have different results. Although most of the CPU execution is deterministic (e.g., `ADD`), non-deterministic instructions (e.g., get user input) are also pervasive. Typical sources of nondeterminism include system calls, interrupts, signals, and data races for concurrency programs[1]. All these non-deterministic events can be futher classified into two types: inconstancy of the data flow - for example, certain system calls such as `getrandom` and `getpid`, and inconstancy of the control flow - for example, concurrency bug due to memory access in inconsistent order[2].

Record-and-replay is a type of approaches that addresses this challenge. Most Record-and-replay systems work by first recording non-deterministic events during the original run of a program and then substituting these records during subsequent re-execution. Record-and-replay system could ultimately guarante that each replay will be identical with the initial version. The fact that a number of replay systems have been built and put into use in recent years illustrates the value of record-and-replay systems in practice[3-6].

There is a rich amount of research on record-and-replay systems, and we can find their various treatments of non-deterministic records. Early record-and-replay systems tend to use virtualization techniques so as to observe and record the entire program non-deterministically on the hypervisior, but the virtual machine is very heavy[7, 8]. Some systems use dynamic binary instrumentation to get the results after running each instruction, but this is very inefficient[6]. There are some other systems that choose not to record at runtime in order to address the expensive cost of recording; instead, they infer these non-deterministic events based on the control flow and other information collected[5, 9]. However, inference often does not reproduce program execution as faithfully as records, and the time required for inference, which in the worst case is a search of the entire space, is a problem[4]. There are also systems that use custom hardware, which inevitably affects its usefulness in practice[10]. Recently there have been some practical systems that have adopted tools provided by Linux for tracing, thus achieving better efficiency. Nevertheless, it still introduces a considerable overhead (50%) and is therefore only available when the developer exactly needs to record and replay[3].

This thesis focuses on the data record part of record-and-replay systems, precisely, the recording of non-deterministic events caused by system calls. I argue that a *practical* record system should (1) run online, meaning that the recording has little performance impact on the execution of the target program, (2) log all data without any omission, (3) work on commercial off-the-shelf hardware, (4) not require any modification to the target program, and also (5) not require any modification to the kernel.

In this thesis, I propose SYSCORD, a practical solution for syscall capturing. It works with unmodified Linux programs on commercial off-the-shelf (OTS) hardware. My original design was on the ARM platform, but the system can be applied to other platforms as well (e.g. x86, riscv). I demonstrat its usefulness on both x86 and ARM platforms. SYSCORD consists of three component: *core hook*, *filter* and *record buffer*.

The *core hook* is a probe of system call. *core hook* inspects each system call, and collects the effects on memory and registers by considering the semantics of system calls. The *filter* stores relevant information of the process what issues the system call, and compares this information with the characteristics specified by the developer. The *record buffer* temporarily store the recording of system calls and dumps it to file.

We implement a prototype of SYSCORD and evaluate it with the aforementioned requirements in mind. The evaluation results show that SYSCORD completely records system calls. We also leverage SYSCORD to record 16 failed programs (7 code segments reconstructed from application and 9 real-world applications including Python, Memcached, and SQLite). The recording indicates that SYSCORD effectively records system calls with a performance overhead of up to 3.88% on average. Meanwhile, SYSCORD directly works on the unmodified binary of the target program and does not rely on any hardware modification.

In summary, I make the following contributions:

- I present a system call recording tool named SYSCORD on Arm platforms, which works with unmodified binary on Arm platform without hardware modification.
- I achieve high performance that allows the always-on trace for the production environment, which provides SYSCORD the ability to reconstruct the entire records.
- I implement a prototype of SYSCORD and evaluate it with real-world applications. The evaluation result demonstrates that SYSCORD successfully records various types of applications with up to 3.88% runtime performance overhead on average.

2. Design

The overarching target of SYSCORD is to capture all syscalls issued by target application with low overhead. Specifically, it has the following characteristics:

- **online:** Our scenario is that our entire record system can work simultaneously on the user side and eventually become a part of the log report. Therefore it must introduce only a extremely low overhead to guarantee that the user can run the desired program without perception.
- **complete:** We also need to ensure the integrity of record results, i.e. that every syscall is correctly captured with all the data needed for reproduction. Not only do we need to verify the correctness of our records on each syscall, but we also have to guarantee that the data in the buffer is fetched in a timely manner without any overflow.
- **off-the-shore:** An practical system should never make assumptions about the hardware. SYSCORD is desgined completely based on the off-the-shelf hardware.
- **without modification to application:** We do not make any changes to the source code or binary of the target application. The entire code of our system runs in kernel space, except for the transfer of logged results that need to be transferred to user space. This means that SYSCORD cares nothing about how the target application in user space is executed, but only about the event that this target application jumps to the kernel state.
- **without modification to kernel:** Modifying Kernel source may make all procedures much more easy. However, this approach will inevitably reduce the compatibility of SYSCORD, especially for devices whose kernels have been modified by manufacturers. Futhermore, to modify kernel, we would also need to synchronize upstream changes. Consequently, SYSCORD is baesd on kernel moudle and works as a loadbale driver.

SYSCORD realizes syscall capturing in three steps. (1) SYSCORD inspect and record each syscall. (2) Next, SYSCORD filter these records by some arrtibutions of its caller process. (3) Finally, SYSCORD transfer these filtered records.

2.1 Desgin Overview

In this section, I present the desgin of SYSCORD by focusing on how it addresses the above two key challenges. SYSCORD contians three parts: *core hook*, *filter*, and *record buffer*.

As Figure 1 shows, in the kernel space, *core hook* collect information for each system call, and then transfer to *filter* part. Subsequently, at the *filter* part, it will find process information from the kernel, and filter syscall records with specific features (e.g., process id or name) and finally pass to *record buffer*. The record buffer manage the buffer in kernel space and file write to user space.

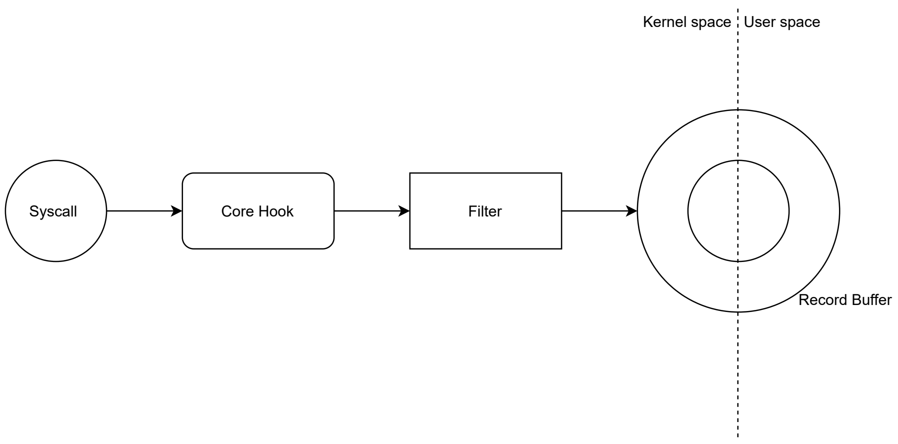

Figure 1: The Overview of Syscord

```
1   // Thread 1::
2   char big_buf[64];
3   while(1)
4     read(fd, big_buf, 64);

5   // Thread 2::
6   int total = 0;
7   int len = 0;
8   char buf[15];
9   for (short i=0; i < 2; ++i)
10    len = strlen(big_buf);
11    if (len < 15)
12      strcpy(buf, big_buf);
13      total += len;
14  assert(total<30)
```

Figure 2: Buffer Overflow Caused by Data Race

2.2 Case Study

Figure 2 demonstrates a typical concurrency bug related to the non-deterministic event (syscall `read`). Assume that the loop (Line 10 to Line 13) in Thread 2 is executed twice. In the first iteration, the `read` of `big_buf` (Line 4) in Thread 1 is performed after the length check (Line 10 and Line 11) in Thread 2. The following `strcpy` (Line 12) in Thread 2 may lead to a buffer overflow and overwrite variables `len` and `total`. In the second iteration, no data race is involved, but the unpredictable `total` overwritten in the first iteration might be larger than 30 after the summation (Line 13) in Thread 2. This finally fails the `assert` (Line 14) in Thread 2 and crashes the program. In this example, the memory and registers indicating the root cause of the bug are overridden by subsequent control flow. Hence, it is necessary to record the content of syscall `read` to figure out the bug.

2.3 Core Hook

Core hook utilizes *Linux Kernel Tracepoints*[12], which provides lightweight hook points in critical positions over the kernel (includes the enter and exit of syscall), to achieve the capture. Figure 3 illustrates the procedure of syscalls (defined at function `el0_svc_`

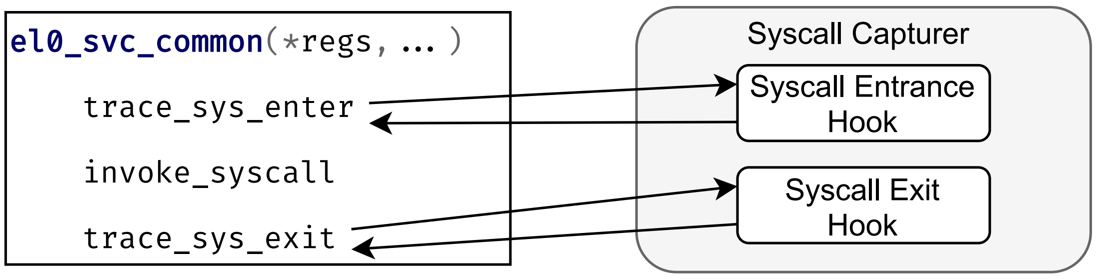

Figure 3: The Two hooks of *core hook*

Table 1: Part of Registers Used in Syscalls[11].

arch	syscall number	return value	return value 2	arg0	arg1	arg2
arm64	w8	x0	x1	x0	x1	x2
x86-64	rax	rax	rdx	rdi	rsi	rdx
riscv	a7	a0	a1	a0	a1	a2

`common` for Arm64) and the recording for syscalls.

When entering a syscall, the kernel goes into *syscall entrance hook* which stores values of involved registers. As Table 1 shows, the register `x0` and `x1` are used as both parameters and return values of syscalls on Arm64. Thus, these registers are overwritten by return values during syscall procedure. Consequently, we may not obtain the syscall parameters at the *syscall exit hook* directly. However, these parameters are critical in some cases. For example, as Figure 2 demonstrates, the content of `big_buf`, addressed by the first parameter of `read`, is necessary for failure analysis. Therefore, we need to record the corresponding parameter at the *syscall entrance hook*, and further record the content addressed by `big_buf` at the *syscall exit hook*.

Before exiting a syscall, the kernel falls into *syscall exit hook*. This hook records different information for various types of syscalls. Firstly, if the capturer notices that a syscall only changes specific registers (e.g., `getpid` only changes return value), it could selectively save information to reduce the size of record file. Secondly, we need to record additional information for certain syscalls; for example, the syscall `read(int fd, void *buf, size_t count)` tries to fill `count` bytes to the memory region addressed by `buf`. In this case, we have to notice and record the write to this memory region.

For a better understanding of the impact of syscalls, we classify them into four types according to their effect on memory and registers.

- **R***eading* **S***tatus*: The *RS-Type* syscalls aim to read information related to system status. The results of these syscalls may be transferred by the return value (e.g., `getpid`), a pointer (e.g., `getitimer`), or shared memory (e.g., `getrandom`). For all syscalls in this category, we can directly record the memory or register they changed.

- **W***riting* **S***tatus*: The WS-Type syscalls changes the status of the system. As the WS-Type syscalls do not directly change the memory and registers of the program, we ignore them unless they fail and return an error code. For example, the syscall `epoll_create` in `epoll` API is ignored, but the influence of syscall `epoll_wait` is recorded since we classify it as an *RS-Type* syscall.

- **R***eading* **C***ontent*: The *RC-Type* syscalls read content from an external input, and the handling of *RC-Type* syscalls is similar to that of *RS-Type* syscalls. However, since the content is usually much larger than the status read in the *RS-Type* syscalls, we choose to truncate the content and record only the first 256 bytes due to performance requirements. Recording the entire content may become impractical in extreme cases (e.g., data center with tremendous uploads) as it results in a huge size of record file. Besides, we find that the truncated content is sufficient for bug diagnosis in most cases. A further evaluation is discussed in §??.

- **W***riting* **C***ontent*: The *WC-Type* syscalls write content to an external source. We consider that they would not affect the execution status of the target program. For example, the syscall `write` is ignored, and the written content is recorded if the *RC-Type* syscall `read` is used to read from the source again.

2.4 Filter

2.5 Record Buffer

The *record buffer* is intended to act as a transit between kernel space and user space. Therefore it has two parts located in kernel space and user space respectively. One of the simplest designs is to maintain a daemon in user space constantly querying and retrieving the data stored in the buffer, and then dumping the data to a file. However, we note that this introduces a huge amount of overhead, mainly due to frequent file io. Placing a larger buffer in user space would also solve this problem, but SYSCORD do not want to introduce a large impact on the entire system.

Therefore, *record buffer* is designed to keep fetching the buffer occupancy, and then dumps the whole buffer only after finding that the buffer occupancy has reached a threshold.

3. Implementation

In this section, we describe our prototype of SYSCORD. We implement SYSCORD with more than `1.6k` lines C/C++ code on an Armv8 board based on Linux. In our prototype of SYSCORD, we analyze the semantics of the 60 commonly used syscalls.

For each syscall, SYSCORD generates a description including return address, process id and its effects. We also use a 16 MB buffer to temporarily store the syscall records and then dump this buffer to files when it is full.

To optimize performance, we further apply a simple encoding rule to reduce the size of record file. The SYSCORD only incurs a low overhead in both time and space. Detailed measurement is presented in §??.

- new sprintf
- encoding

3.1 Core Hook

The part of *core hook* aims to hook system call, i.e., inject custom code at the begin and the end of a system call. By leveraging *core hook*, It is easy to get the return value of the system call and the changes to the parameters. Linux has provided many different approaches to achieve it, such as *ptrace*, *auditd*, *Kprobe* and *tracepoint*.

I choose *tracepoint* to implement our core hook, since it

3.2 Filter

We get the process descriptor of syscall issuer (the process using the syscall) via `current`, and compare it with conditions passed in.

3.3 Record buffer

The buffer is a circular queue. And there is also a number indicating the usage of the buffer.

The most sophisticated component is to share the buffer between kernel space to user space.

4. Evaluation

In this section, we first use a case study (§4.1) given in Figure 2 to show the complete workflow of the SYSCORD. Next, we focus on evaluate SYSCORD with the four practical requirements discussed in §1. Specifically, we aim to answer the following research questions:

RQ 1: How completely can SYSCORD record system calls? (§4.2)

RQ 2: What is the performance of SYSCORD? (§4.4)

RQ 3: Could SYSCORD perform bug analysis on binary code? (§4.5)

We deploy the SYSCORD on an Armv8 Juno r2 board equipped with 6 cores (2 Cortex-A72 cores and 4 Cortex-A53 cores) and 8GB RAM based on Linaro deliverables

Table 2: Tailored Control Flow And Data Flow for Figure 2

Block	Thread 1	Thread 2	Data Values
A		bl 400910 <strlen@plt>	total=0, len=0, buf=?
B	bl 400960 <read@plt>		read, fd=3, size=64, res=25, data="1234567890123456789012345"
C		bl 400970 <strcpy@plt>	total=0, len=875770417, buf= "123456789012345"
D	bl 400960 <read@plt>		read, fd=4, size=64, res=6, data="123456"
E		bl 400910 <strlen@plt>	
F		bl 400970 <strcpy@plt>	total=875770423, len=6, buf="123456"
G		bl 400940 <___assert_fail@plt>	

Linux 5.4.50. We equip the Juno board with an SSD and allocate 256 MB circular buffers for ETM tracing. We use this as the default setting for our experiment but also allowed developers to adjust it as required.

4.1 Case Study

To evaluate the functionality of Syscord, we test it with the program shown in Figure 2. As Table 2 illustrates, it is clear that Thread 2 checks the length of `big_buf` (Line 10, Block 2-A) before the first `read` (Line 4, Block 1-B) in Thread 1. Since we capture the data of `read`, we know that the `big_buf` contains a string with length 25. Therefore, the following `strcpy` (Line 12, Block 2-C) incurs a buffer overflow, and the variable `len` (Line 7) is unexpectedly changed to `875770417`, which finally leads to the failure of the `assert` (Line 14, Block 2-G).

It is notable that the program does not crash immediately after the buffer overflow. In contrast, it executes normally for the second cycle (Block D to F). In addition, the normal execution of the second time overwrites the values (`buf` and `len`) involved in the buffer overflow, which may confuse developers without the assistance of Syscord.

4.2 Completeness

4.3 Effectiveness

We show how effectively Syscord is for diagnosing the root cause of bugs. As listed in Table 3, we use 16 commonly C/C++ buggy programs[13-18] to evaluate Syscord. We divide these bugs into two groups, i.e., Group E and Group R. Group E contains 7 bugs reconstructed from applications[15, 16], and Group R includes 9 bugs in real-world applications[13, 14, 17, 18]. There are 13 concurrency bugs, of which 6 are single variable atomicity violation (SAV), 4 are multi variables atomicity violation (MAV), 2 are deadlock (DL), and 1 is order violation (OV). There are also 3 non-concurrency bugs. These bugs are collected from a diverse set of real-world systems (e.g., Python, Memcached, SQLite, and Aget) and wide symptoms (e.g., NULL pointer dereference, use-after-free, and race).

We execute these programs separately in our system until the bug occurs and then use Syscord to analyze the bug. We receive the identified root cause from the

Table 3: Syscalls issued from bugs recorded by SYSCORD. N/A=bugid is not available, E=reconstructed bug, R=real-world bug, OV=order violaion, SAV=single variable atomicity violation, MAV=multi variables atomicity violation, DL=deadlock, SEQ=sequential bug (non-concurrency bug), LOC=line of code

Program-BugID-GroupType	**bug type**	**LOC**	**Symptom**
shared_counter-N/A-E	SAV	45	assertion failure
log_proc_sweep-N/A-E	SAV	93	segmentation fault
bank_account-N/A-E	SAV	95	race condition fault
jdk1.4_StringBuffer-N/A-E	SAV	180	assertion failure
circular_list-N/A-E	MAV	155	race condition fault
mysql-169-E	MAV	120	assertion failure
mutex_lock-N/A-E	DL	51	deadlock
SQLite-1672-R	DL	80K	deadlock
memcached-127-R	SAV	18K	race condition fault
Python-35185-R	SAV	1256K	race condition fault
Python-31530-R	MAV	1256K	segmentation fault
aget-N/A-R	MAV	2.5K	assertion failure
pbzip2-N/A-R	OV	2K	use-after-free
curl-965-R	SEQ	160K	unhandled input pattern
cppcheck-2782-R	SEQ	120K	unhandled input pattern
cppcheck-3238-R	SEQ	138K	NULL pointer dereference

SYSCORD and confirm that it is effective for root cause analysis of all the 16 bugs. Specifically, we manually analyze and compare the related patches of these bugs. The result indicates that the failure reports generated by SYSCORD are exactly related to the root cause. In all these programs, the timestamps generated by ETM are precise enough to determine the accurate control and data flow that caused the failures, which allow us to diagnose the bugs and reproduce all of them successfully.

Out of those 16 bugs, we select 3 representative examples to further demonstrate the effectiveness of SYSCORD.

Cppcheck-2782. In this case, we use SYSCORD to analyze a non-concurrency

Table 4: SYSCORD output of Cppcheck-2782

PID	**Syscall**	**Parameters**	**Additional information**
22571	getcwd	-	path=/home/root/cppcheck
22571	newfstatat	res=0	-
22571	fstat	res=0	fd=1
22571	write	res=29	-
22571	openat	res=3	dir=-100, path=./fail.cpp
22571	read	fd=3	data="int main() { return0; } #asm !while (val) mov bx #endasm"
22571	close	fd=3	res=0

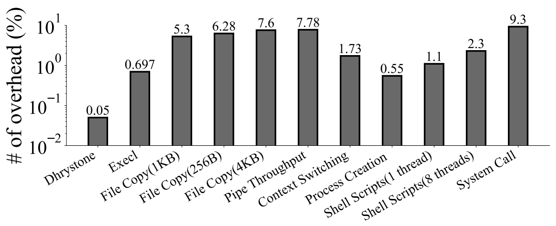

Figure 4: Performance overhead of running UnixBench with ETM tracing and non-deterministic event TheName.

bug. We run the application with common C++ source code as its input until it crashes. With the help of ETM tracing, the Control Flow Builder gives us a complete execution history of the target program from the beginning to the crash point. We can get the crash function `Tokenizer::removeMacrosInGlobalScope()` by comparing the symbols in the recorded control flow. Not only can SYSCORD provide the crash function point, it also helps us further analyze the root cause of the crash. The Data Flow Builder restores the memory and register information, indicating the crash is because illegal memory access. Further more, by checking the output from non-deterministic event TheName, we can also examine the user input to find the file which triggers the bug. Table 4 shows the syscalls recorded by the non-deterministic event TheName. The C++ source code analyzed by Cppcheck which triggered the bug is loaded with the syscall `read`. We then find the source code is special for containing embedded assembly code. According to the public bug report, the Cppcheck-2782 is known as a bug caused by unhandled input pattern, for its disability in handling embedded assembly code. SYSCORD can correctly locate the bug and easily analyze the root cause.

In contrast to the state-of-the-art system (REPT), we can support the detection of failures caused by non-deterministic events (e.g., curl-965, cppcheck-148, and cppcheck-3238). Lack of handling online data prevents REPT from recovering crucial data, making it cannot detect these bugs.

4.4 Efficiency

We show how efficiently SYSCORD can be used for bug analysis by first running Unixbench 5.1.2[19] to measure the performance impact on kernel such as syscall. We then run ApacheBench[20] with Nginx 1.20[21], representing a popular server program to simulate a high load scenario. We finally evaluate the runtime performance overhead of SYSCORD by running four real-world programs.

4.4.1 Unixbench

We run UnixBench on Linux and show the performance results in Figure 4. For the tracing enabling, the performance overhead is 3.88% on average, and the highest

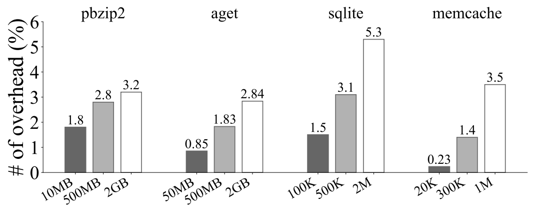

Figure 5: Performance overhead of 4 real-world prgrams execution enables ETM tracing and non-deterministic event TheName with library hook.

performance overhead is for System Call at 9.3%. Specifically, there are three types of benchmarks: File Copy, Pipe Throughput, and System Call that have higher overhead. We believe that it is because these benchmarks have more frequent syscall invoking and I/O operations, which incur larger overhead than others.

4.4.2 Nginx

We use `nginx`[21] as a web server program to test the performance of SYSCORD in a high concurrency environment. We use `ab` (Apache HTTP server benchmarking tool)[20] to simulate user access behavior. Setting `nginx` to its default configuration, we operate performance testing with concurrency of 5,000 and request number of 500,000.

The average time cost for baseline (i.e., without SYSCORD) is 88.94s, and SYSCORD is 90.09s with 1.30% overhead. This shows that SYSCORD performs well even in high-pressure environments.

4.4.3 Performance overhead on real-world programs

We use four real-world programs to test the SYSCORD for performance overhead of normal executions, including `Pbzip2`, `Aget`, `SQLite`, and `Memcached`. We run different fine-grained tests on each of the programs to simulate three different load scenarios. We run `Pbzip2` to compress 10MB, 500MB, and 2GB files, respectively. We use `Aget` to download 50MB, 500MB, and 2GB files in the same network to avoid network speed interference. `SQLite` is evaluated by a `sqlite-bench`[22] to write $100,000$, $500,000$, and $2,000,000$ values in sequential key order in sync mode, respectively. A benchmark tool `Twemperf`[23] was used to test `Memcached`, which creates $20,000$, $300,000$, and $1,000,000$ connections to a Memcached server running on localhost.

We show performance overhead results in Figure 5. Overall, the average performance overhead of all tests is 2.3%, and the highest overhead is 5.3% for `SQLite` when writing $2,000,000$ values. The performance overhead has a slight improvement when test stress increases in all four programs. We believe it is caused by SYSCORD because there are many I/O operations and a large number of syscalls.

We compare the runtime performance overhead of real-world programs with the state-of-the-art systems[13, 14]. The results show that our overhead is slightly higher,

Table 5: Space Consumption for Saving All Content or Truncation

Type	**Real Time**	**File Size**	**Estimated 24-hour File Size**
Baseline	2 min 50.3 s	-	-
All Content	3 min 11.552 s (+12.5%)	2.0 GB	902.1 GB
Truncation	2 min 57.675 s (+4.33%)	120 MB	52 GB

mainly caused by the SYSCORD. Nevertheless, SYSCORD still incurs a low runtime performance overhead (2.3% on average). We conclude this overhead is acceptable and still suitable for practical deployment.

4.4.4 Trade-offs Between Performance and Accuracy

We design two versions of syscall capturing to record *RC-Type* syscalls: saving the whole content or truncating to the beginning 256 bytes . We test the time and space consumption of both versions by a program that continuously reads a 2 GB text file to simulate a highly concurrent environment on the server. The experimental results shown in Table 5 demonstrate that saving the entire content imposes a significant overhead (12.5%). In addition, an estimate for 24 hours of continuous recording generates nearly 1 TB files, which indicates that saving the entire content is also impractical for a larger throughput server. Therefore, we choose to truncate the records of *RC-Type* syscalls to the beginning 256 bytes.

4.5 Generality

5. Related Work

There is a large amount of related work dedicated to cpaturing syscalls. In this section, I discuss some representative examples and describe how SYSCORD differs.

- **Pinplay** ...
- **REPT** ...
- **rr** ...
- **DTrace** ...
- **sysdig** ...

Bibliography

[1] RONSSE M, DE BOSSCHERE K. RecPlay: a fully integrated practical record/replay system[J]. ACM Transactions on Computer Systems, 1999.

[2] MICHAEL H, DENYS V. Getrandom(2) —Linux manual page[EB/OL]. 2021 [2021-03-27]. https://man7.org/linux/man-pages/man2/ptrace.2.html.

[3] O' CALLAHAN R, JONES C, FROYD N, et al. Engineering Record and Replay for Deployability[C]//2017 USENIX Annual Technical Conference (USENIX ATC 17). Santa Clara, CA: USENIX Association, 2017: 377-389.

[4] CHEN Y, ZHANG S, GUO Q, et al. Deterministic Replay: A Survey[J]. ACM Comput. Surv., 2015, 48(2). DOI: 10.1145/2790077.

[5] ALTEKAR G, STOICA I. ODR: output-deterministic replay for multicore debugging[C]//Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09. Big Sky, Montana, USA: ACM Press, 2009: 193 [2021-03-21]. DOI: 10.1145/1629575.1629594.

[6] BHANSALI S, CHEN W K, de JONG S, et al. Framework for instruction-level tracing and analysis of program executions[C]//VEE '06: Proceedings of the 2nd international conference on Virtual execution environments. New York, NY, USA: Association for Computing Machinery, 2006: 154-163.

[7] DUNLAP G W, KING S T, CINAR S, et al. ReVirt: enabling intrusion analysis through virtual-machine logging and replay[J]. ACM SIGOPS Operating Systems Review, 2003, 36(SI): 211-224.

[8] DUNLAP G W, LUCCHETTI D G, FETTERMAN M A, et al. Execution Replay of Multiprocessor Virtual Machines[C]//VEE '08: Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. Seattle, WA, USA: Association for Computing Machinery, 2008: 121-130. DOI: 10.1145/1346256.1346273.

[9] CUI W, GE X, KASIKCI B, et al. REPT: Reverse Debugging of Failures in Deployed Software[C]//OSDI'18: Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation. Carlsbad, CA, USA: USENIX Association, 2018: 17-32.

[10] MONTESINOS P, HICKS M, KING S T, et al. Capo: A Software-Hardware Interface for Practical Deterministic Multiprocessor Replay[J]. SIGARCH Comput. Archit. News, 2009, 37(1): 73-84. DOI: 10.1145/2528521.1508254.

[11] University of California. Syscall(2) - Linux manual page[EB/OL]. 2021 [2021-03-28]. https://man7.org/linux/man-pages/man2/syscall.2.html.

[12] MATHIEU D. Using the Linux Kernel Tracepoints —The Linux Kernel documentation[EB/OL]. 2021. https://www.kernel.org/doc/html/latest/trace/tracepoints.html.

[13] CUI W, GE X, KASIKCI B, et al. REPT: Reverse debugging of failures in deployed software[C]//13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18). [S.l. : s.n.], 2018.

[14] KASIKCI B, CUI W, GE X, et al. Lazy Diagnosis of In-Production Concurrency Bugs[C]//Proceedings of the 26th Symposium on Operating Systems Principles. Shanghai China: ACM, 2017: 582-598.

[15] YU J, NARAYANASAMY S. A case for an interleaving constrained shared-memory multi-processor[J]. ACM SIGARCH Computer Architecture News, 2009.

[16] YU J, NARAYANASAMY S, PEREIRA C, et al. Maple: A coverage-driven testing tool for multithreaded programs[C]//Proceedings of the ACM international conference on Object oriented programming systems languages and applications. [S.l. : s.n.], 2012.

[17] KASIKCI B, SCHUBERT B, PEREIRA C, et al. Failure sketching: A technique for automated root cause diagnosis of in-production failures[C]//Proceedings of the 25th Symposium on Operating Systems Principles. [S.l. : s.n.], 2015.

[18] LIANG J, LI G, ZHANG C, et al. RIPT–An Efficient Multi-Core Record-Replay System[C]//Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. [S.l. : s.n.], 2020.

[19] Kdlucas. Byte-unixbench[EB/OL]. 2018. https://github.com/kdlucas/byte-unixbench.

[20] The Apache Software Foundation. Apache HTTP server benchmarking tool[EB/OL]. 2018. https://httpd.apache.org/docs/2.2/programs/ab.html.

[21] NGINX. Nginx-1.20.0 stable version[EB/OL]. 2021. http://nginx.org/download/nginx-1.20.0.tar.gz.

[22] Retrage. SQLite3 Benchmark[EB/OL]. 2018. https://github.com/ukontainer/sqlite-bench.

[23] Thinkingfish. Twemperf[EB/OL]. 2014. https://github.com/twitter-archive/twemperf.