System Call Trace Record/Replay Project

Shubhi Rani - Stony Brook University
Nina Brown - Harvey Mudd College
July 26, 2016

Agenda

- Introduction
- Workflow
- Trace Recording
  - Strace
  - Strace2ds library
- Trace Replaying
- System Calls Supported
- Replayer Results
- Changes to SNIA document
- Issues not yet addressed
- What's next?

Introduction

Overview

- Tracing and Replaying tool
- Intended for
  - extracting system call traces from running any program or application
  - replaying their I/O behavior
- Uses DataSeries
  - Fast and efficient format in which to store traces
- Follows SNIA specification documents
  - POSIX System-Call Trace Common Semantics
  - I/O Trace Common Semantics

Motivation

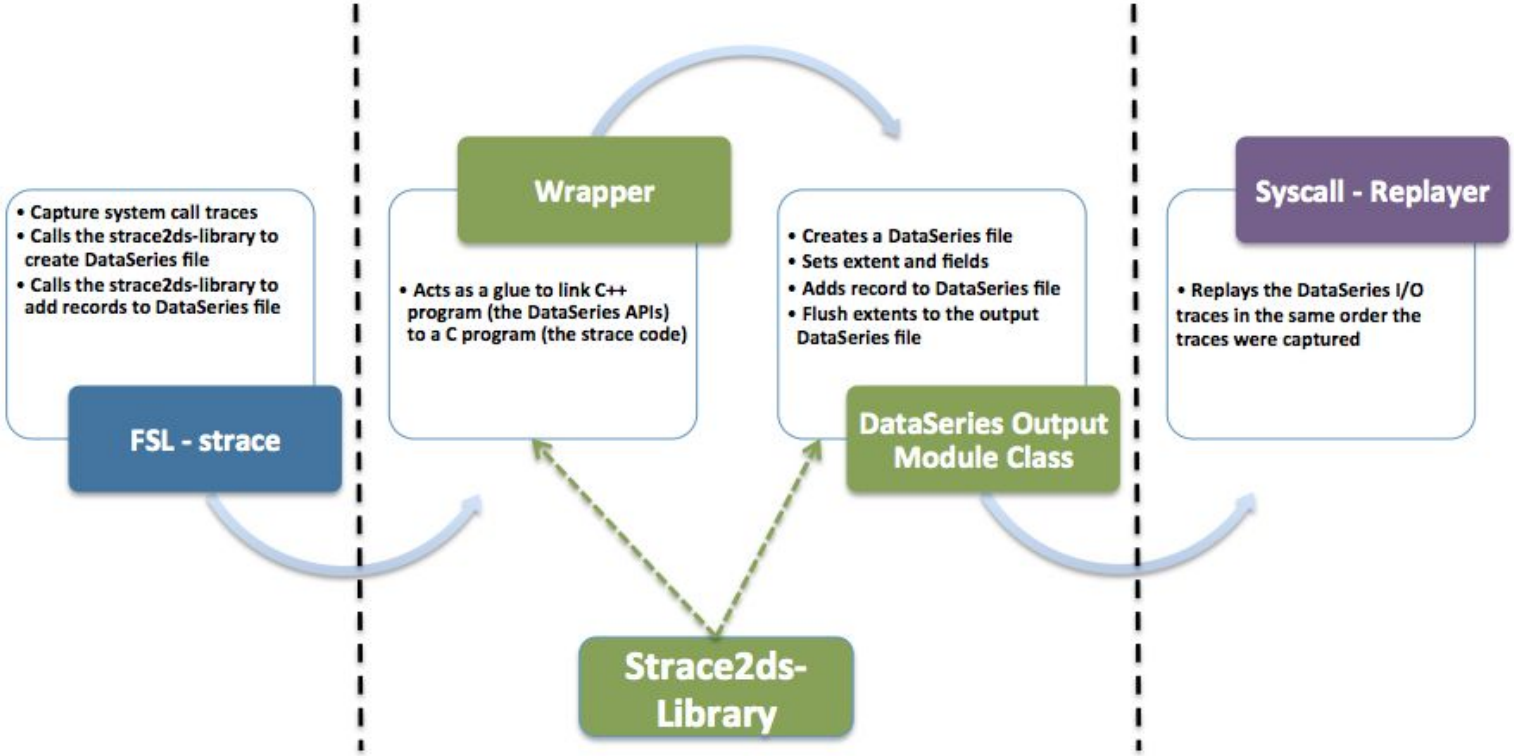
- To be able to accurately record and replay workloads using Linux system calls

- Possible uses:
  - Simulate I/O behavior of an application or a program
  - Benchmark operating system features or alterations
  - Analyze program security and other characteristics

- Ultimate goal:
  - Record and replay large server apps (i.e. apache, mysql)

Workflow

High - level diagram



Trace Recording (FSL - strace)

Approach

- Minimize changes to strace code
- Every change is wrapped in #ifdef ENABLE_DATASERIES
  - Enabled with ./configure --enable-dataseries
- Run-time option "-X *ds-filename*" to enable functionality
- When writing to a DataSeries file, strace's normal (human-readable) output is turned off
  - Prints warning message if an unsupported system call is traced
- strace2ds-library also uses autoconf

Why this approach?

- Previous attempt failed
  - Parse strace's human-readable output and convert it to a CSV file
  - Convert the CSV file to a DataSeries file
  - Issues
    - Parsing the output of certain system calls was very difficult (e.g., stat)
    - Required multiple distinct steps
    - Binary information (buffer contents) not always available
- This new approach
  - Easy to run
  - We have access to all of the information strace collects about each traced system call
  - Uses strace's code to our advantage

Trace Control Block (Strace: struct tcb)

```
/* Trace Control Block */
struct tcb {
        int flags;              /* See below for TCB_ values */
        int pid;                /* If 0, this tcb is free */
        int qual_flg;           /* qual_flags[scno] or DEFAULT_QUAL_FLAGS + RAW */
        int u_error;            /* Error code */
        long scno;              /* System call number */
        long u_arg[MAX_ARGS];   /* System call arguments */
#if defined(LINUX_MIPSN32) || defined(X32)
        long long ext_arg[MAX_ARGS];
        long long u_lrval;      /* long long return value */
#endif
        long u_rval;            /* Return value */
#if SUPPORTED_PERSONALITIES > 1
        unsigned int currpers;  /* Personality at the time of scno update */
#endif
        int sys_func_rval;      /* Syscall entry parser's return value */
        int curcol;             /* Output column for this process */
        FILE *outf;             /* Output file for this process */
        const char *auxstr;     /* Auxiliary info from syscall (see RVAL_STR) */
        const struct_sysent *s_ent; /* sysent[scno] or dummy struct for bad scno */
        const struct_sysent *s_prev_ent; /* for "resuming interrupted SYSCALL" msg */
        struct timeval stime;   /* System time usage as of last process wait */
        struct timeval dtime;   /* Delta for system time usage */
        struct timeval etime;   /* Syscall entry time */

#ifdef USE_LIBUNWIND
        struct UPT_info* libunwind_ui;
        struct mmap_cache_t* mmap_cache;
        unsigned int mmap_cache_size;
        unsigned int mmap_cache_generation;
        struct queue_t* queue;
#endif
};
```

Strace: Common Fields and Virtual Arguments

- Void * arrays
- Used to pass system call arguments, information to strace2ds-library functions
- Common Fields:
  - Time called, time returned, return value, errno number, pid
- Virtual Arguments (v_args):
  - Pathnames, read/write buffers, struct stat, etc. as needed
  - Memory copied from the traced process's address space to strace's address space

Strace: Utility functions

- ds_get_* (path, buffer, etc.)
- Located in util.c
- These functions call umoven() or umovestr() which are strace's own utility functions
  - Copy data from a traced process's address space to strace's address space

Strace : Utility function example

```
void *
ds_get_buffer(struct tcb *tcp, long addr, long len)
{
        void *buf = NULL;

        if (!addr || len < 0)
                goto out;

        /*
         * Note: xmalloc succeeds always or aborts the trace process
         * with an error message to stderr.
         */
        buf = xmalloc(len);

        if (umoven(tcp, addr, len, buf) >= 0)
                goto out; /* Success condition */

        if (buf) {
                free(buf);
                buf = NULL;
        }
out:
        return buf;
}
```

How We Get Information from Strace: syscall.c

- SEN_*: strace's own identification numbers for system calls
  - Ex: SEN_open, SEN_close
- Primarily in trace_syscall_exiting(), an strace function called just after a system call is executed
- Switch statement with a case for each supported system call
  - Switch block is defined in trace_syscall_exiting() function
  - In each case, get any virtual arguments and call ds_write_record(), an strace2ds-library function, to add a record in the DataSeries output file
- Non-terminating system calls (i.e., exit, execve) are recorded from trace_syscall_entering(), which is called just before a system call is executed

Simple Use Case for trace recording

```
switch (tcp->s_ent->sen) {
        case SEN_open: /* Open system call */
                v_args[0] = ds_get_path(tcp, tcp->u_arg[0]);
                ds_write_record(ds_module, "open", tcp->u_arg,
                                common_fields, v_args);
                break;
```

- tcp->s_ent->sen : unique for each system call (SEN_* numbers)
- ds_get_path(): returns the actual pathname passed to open system call.
- ds_write_record(): adds the captured trace to the DataSeries file.

Trace Recording (strace2ds-library)

Fields Table and XML Files

- Table format
  - Delimited by '\t'
  - Four fields
    - Extent (system call) name
    - Field name
    - Nullable (1) or non-nullable (0)
    - Field type (int32, int64, bool, byte, variable32)
- XML files
  - One for each system call
  - Created using script generate-xml.sh <table-filename>
  - Table and XML must exist prior to running strace -X <ds-filename>

Wrapper code (strace2ds.h, strace2ds.cpp)

- Three major functions:
  - ds_create_module():
    - Calls DataSeriesOutputModule constructor to create a new DataSeries file
    - Called once when strace starts
  - ds_write_record():
    - Calls DataSeries APIs to add new records in the DataSeries file
    - Called each time a (supported) system call is traced.
  - ds_destroy_module():
    - Calls DataSeriesOutputModule destructor to flush all the records to the DataSeries output file.
    - Called once when strace ends

DataSeriesOutputModule Class

- One DataSeriesOutputModule object is created per trace
- When initialized, creates configuration table from the XML files with extent names and the relevant fields
- Variables
  - OutputModuleMap modules_
    - map<extent name, OutputModule>
  - ExtentMap extents_
    - map<extent name, FieldMap>
      - FieldMap: map<field name, <DS field, DS field type>>
  - DataSeriesSink
    - Wrapper for DataSeries output file
  - config_table_
  - u_int record_num

DataSeriesOutputModule : Major functions

- Three major functions
  - Constructor():
    - Initializes configuration table (tables/snia_syscall_fields.table)
    - Register extent types to the library
    - Loop through each extent to
      - Create its fields from xmls
      - Create Extent series, Output Module and fields
  - writeRecord():
    - Add a new record to the DataSeries file
    - Register the record and field values into DataSeries fields
  - ~Destructor():
    - Flush all the records to the output file
    - Destroy the DataSeriesOutputModule object

DataSeriesOutputModule : Helper functions

- Three major helper functions
  - addExtent()
    - Add a new extent (system call) to the DataSeries file
  - setField()
    - Sets the DataSeries field with its corresponding values
  - make[system call name]ArgsMap(....)
    - Creates a mapping between system call field names and their corresponding values
    - This map is used later, with setField(), to set each field in the DataSeries record

Workflow

- Run strace with ‘-X’ option
- Each system call is caught inside switch block
- From switch block, call ds_write_record() [wrapper code]
- ds_write_record() calls DataSeriesOutputModule::writeRecord()
  - Creates map<string, void *> of field names to field values
  - Stores the common fields in the map if present
  - Checks if the system call name matches a supported system call
  - Make[system call name]ArgsMap (...)
    - Stores system call specific arguments in the map
  - Iterate through field names, set values, and write to DataSeries record

How to run strace

strace [-X *ds_filename*] program_name

Trace Replaying

Base SystemCallTraceReplayModule Class

```
class SystemCallTraceReplayModule : public RowAnalysisModule {
protected:
  std::string sys_call_name_;
  bool verbose_;
  int  warn_level_;
  Int64Field time_called_;
  Int64Field time_returned_;
  Int64Field time_recorded_;
  Int32Field executing_pid_;
  Int32Field errno_number_;
  Int64Field return_value_;
  Int64Field unique_id_;
  int rows_per_call_; // It stores the number of rows processed per system call.
  int replayed_ret_val_;
```

Individual System Call Module Classes

- Each system call has its own module class derived from SystemCallTraceReplayModule
- Variables
  - arguments specific to that system call
- Methods
  - print_specific_fields()
  - processRow()

Priority Queue

- Min heap is defined that stores each system call module.

- Modules are processed in the order of minimum unique_id number

Workflow

- Initialize a module object for each supported system call
- Replay system calls in order of unique_id number
  - processRow() defined in each system call's module class
    - Gets argument values from the ExtentSeries
    - Actually replays the system call
  - completeProcessing()
    - Compares traced and replayed return values and errno numbers
    - Prints system call arguments if in verbose mode
    - Adjusts location in ExtentSeries appropriately

Replayer Design Decisions - I

- File Descriptor Map
  - System calls may not replay with the same file descriptors as were traced
    - Maintain a map<int, int> of traced file descriptor to replayed file descriptor
    - Map certain standard values (stdin, stdout, stderr, AT_FDCWD) to themselves before replay begins

- Integer encoding of flag/mode arguments
  - The SNIA document specifies that individual flags/modes bit should be recorded as boolean fields, so we do record them
  - However, it is faster when replaying a trace to simply pass the traced integer encoding to the system call

Replayer Design Decisions - II

- Rows per call
  - Most system calls can be fully encoded in a single record and take up one row in an extent
  - rows_per_call_ is a member variable of the base SystemCallTraceReplayModule class
  - The default value of the rows_per_call_ variable is 1
  - Some require more than one record (e.g., readv, writev, execve)
  - For these system calls, rows_per_call_ is set accordingly in processRow()

- It doesn't make sense to replay some system calls
  - E.g., _exit, execve, mmap
  - We record them in trace, and identify that they were traced when replaying, but don't actually replay them

How to run replayer

system-call-replayer [-vV] [--verify] [-p *ARG*] [-w *N*] ds_filename

Replaying Options - I

- Default (-w 0)
  - Prints a message when the first and last system calls of a certain type are prepared/replayed
- Warn (-w 1)
  - Prints a warning message if traced and replayed return value/errno number aren't the same
  - Default: no warning
- Abort (-w 2)
  - Aborts replaying if the traced and replayed return value/errno number aren't the same
  - Default: don't abort
- Version (-V)
  - Prints the version of the syscall-replayer
  - Default: does not print anything
- Verbose (-v)
  - Prints each system call's common fields and arguments
  - Default: not verbose

Replaying Options - II

- Verify (--verify)
  - Verifies that traced and replayed data in read/write buffers, struct stat, etc. is the same.
  - Default: no verify
  - Warn mode: displays contents of both traced and replayed data and continue to replay
  - Abort mode: displays contents of both traced and replayed data and then aborts the replayer program
- Write Pattern data (-p *ARG*)
  - If ARG is specified, fills write buffers with
    - 0: write zeros, e.g., -p 0 (default mode)
    - *pattern*: write a repeated patterns (e.g., -p 0x5)
    - random: generate random data using rand()
    - urandom: generate random data from /dev/urandom
- Multiple options can be used at the same time
  - Abort mode and warn mode cannot be used together

System Calls Supported

List of Supported System Calls

1. open
2. openat
3. close
4. read
5. pread
6. readv
7. write
8. pwrite
9. writev
10. lseek
11. mknod
12. stat
13. fstat
14. lstat
15. fcntl
16. access
17. truncate
18. chdir
19. mkdir
20. rmdir
21. creat
22. link
23. unlink
24. unlinkat
25. symlink
26. readlink
27. chmod
28. chown
29. fsync
30. utime
31. utimes
32. dup
33. dup2
34. pipe
35. rename
36. getdents
37. execve
38. _exit
39. mmap
40. munmap

Not in original SNIA document
Non-replayed system call

Replayer Results

Utilities Replayed Successfully

- cp
- mv
- rm, rm -r
- ls
- mkdir

SNIA document

Suggested Changes to SNIA Posix System Call Trace Semantics Document

- All changes are redlined in the .doc file
- Added system calls rename, getdents, openat, unlinkat
- Added/removed fields due to relevance/redundancy
- Made certain fields nullable or non-nullable
- Fixed typos and inconsistencies

Issues Not Yet Addressed

Testing Protocol

- We've been testing individual system calls with simple test programs centered around one system call
  - Some tests depend on others (e.g., read(2) needs open(2))
- To do:
  - Design a test script aligned with current approach to replay system calls.

Necessary Statistics

- We will need to add support to collect statistics about trace replay
  - For benchmarking the replayer itself
  - For the replayer to be used as a benchmarking tool

- Ex: Timing replayed system calls

Unsupported System Calls

- mremap, mprotect, madvise, mlock, msync
- exec, execl, etc.
- clone, fork
- ioctl
  - Work in progress: hard to tell how many bytes to record by ioctl type

Trouble Replaying ls -l

- When the command ls -l is used, a socket is created and connected to /var/run/nscd/socket to look up uid/gid values

- Since we aren't tracing or replaying socket() and related system calls, other system calls that rely on the existence of the socket (i.e., some read and close calls) fail upon replay

What’s next?


STONY BROOK
ESTABLISHED
STATE UNIVERSITY OF NEW YORK
1957
HARVEY
MUDD
COLLEGE

Replay Larger Applications

- Our strace will warn us if it traces a system call we don't support
  - We can add support for more system calls as needed
- Replay bigger and bigger applications
  - E.g., /bin/ls and /bin/cp in various modes
- Ultimate goal
  - Replay server apps (mysql, apache, etc.)
  - Focus on storage and file system replaying first
- Submit a paper (where/when?)
- Release code (SNIA?)
  - How to release updated spec to SNIA?

Questions ?


STONY BROOK
STATE UNIVERSITY OF NEW YORK
1957
HARVEY
MUDD
COLLEGE