Deterministic replay of program execution based on Valgrind framework.

Research-in-progress report
Maksim Ryndin
Moscow Institute of Physics and Technology, Dolgoprudny, Russian Federation
Scientific supervisor: S. S. Gaisaryan, Institute for System Programming, Moscow, Russian Federation

***Abstract*—Deterministic replay is execution of the same sequence of instructions with the same arguments as at the first run.**

Developers spend a lot of time in trying to fix bugs in their software. Sometimes it is very difficult even to reproduce the bug. This often occurs in multithreaded applications. According to NIST[1] errors in software are estimated $59.5 billion annually. Therefore deterministic replay of program execution has become a problem of interests of many researches recently. Deterministic replay could help to solve the problem of bug reproduction and provide more abilities for program analysis.

Valgrind is a framework for dynamic binary analysis. It provides infrastructure for translation and instrumentation of executable code.

The goal of the work is to create a tool providing deterministic replay of program execution based on Valgrind infrastructure.

I. Introduction

Application programs are widely used in modern world: for automation of business processes, for scientific calculations, for processing large amount of data, etc. The programs become more and more complex and large to provide all needs of people they are used by. The larger the program the more difficult to implement it without errors. So bugs occur quite often and it is necessary to be able to find and fix them quickly.

Finding of bugs' causes could be complicated by unreproducible behavior. Such errors are the most painful for developers and can take a lot of time to fix them.

Multi-core processors become the norm recently. So there is an interest in writing parallel programs. Such programs are more efficient for wide class of applications. Threads in one program must be synchronized in a proper way to prevent race conditions. But it is difficult to guarantee correct synchronization and threads in real programs behave differently from run to run. Often such differences are invisible and never cause errors. But sometimes they may cause or may not cause an error depending of some circumstances. In that case it becomes unreproducible bug.

By deterministic replay one should mean execution of the same sequence of instructions with the same arguments as at the first run. Deterministic replay would provide ability to reproduce bugs at each run.

Several generic dynamic binary instrumentation frameworks exist, such as Pin[2], DynamoRIO[3] and Valgrind[4][5]. These tools make possible to detect variety of errors. E.g. Valgrind distribution includes the following debugging and profiling tools: Memcheck, Cachegrind, Massif, Helgrind, DRD[6]. Memcheck detects wrong memory accesses (e.g. when accessed region of memory is not allocated), usage of uninitialised values, memory leakages, etc. Cachegrind profiles the cache, Massif profiles the heap. Helgrind and DRD both detect race conditions with some differences.

Pin is a proprietary program and it is free for non-commercial use only [7]. Copyright restrictions may also complicate code modifications for research goals.

All these frameworks, however, do not provide deterministic replay of program execution.

BugNet[8][9] and PinPlay[10] both provide deterministic replay. However, at the moment of writing this paper author couldn't find neither source code nor executables of any of them. PinPlay is a proprietary program too, so it would have all license limitations of Pin.

II. Existing tools

Several research papers describing architecture of BugNet and PinPlay are available.

A. BugNet

BugNet is based on the observation that the following information is sufficient to replay a program's execution in a deterministic way: initial architecture state (program counter and register values), architecture state updates from system calls and interrupts and used load values[8].

Architecture state can be kept in a consistent state by recording of register values and program counter after servicing system calls and interruptions.

Logging of each memory load causes overheads, so BugNet logs only the first access to a memory location. To accomplish that the tool marks logged memory addresses. Taking into account the external updates of logged memory such as system calls, DMA and shared-memory interactions, BugNet unmarks corresponding location's address when such events occur. Than it logs again when the location is accessed next time.

BugNet breaks a thread's execution into checkpoint intervals. For a checkpoint interval BugNet stores enough information to start replaying program's execution from the start of the checkpoint. An interval is terminated by system call, interrupt, context switch or when size of stored data exceeds some predefined value.

In addition to data, BugNet records information about the code executed during logging. The information contains name and path of the binary or library loaded, a checksum to represent the version of the binary or library and the starting address where it was loaded. This information goes to *code log*.

Replay of a checkpoint interval starts with reading the code log and restoring of code space. Then architecture state (program counter and register values) is set. Then executions starts. For every load instruction the replayer obtains the load value from load-log if the value was not obtained earlier. At the end of the checkpoint interval a new one starts.

B. PinPlay

PinPlay consists of two Pintools[10]: a *logger* and a *replayer*. The logger stores information about program execution in a set of files called *pinball*. The replayer repeats program execution using information from pinball.

A very powerful feature of PinPlay is ability to combine the logger and replayer with most existing Pintools and GDB. It gives lots of useful information about a bug and helps fixing it easily.

The PinPlay logger stores only minimal information necessary to reproduce the non-deterministic events and thus does not generate too large pinballs. The sources of non-determinism are:

- *Initial stack location*: the location is assigned by the kernel and may differ from run to run
- *Data location changes*: addresses of allocated memory may also differ
- *Program code changes*: code of shared libraries may change from machine to machine
- *CPU specific instruction behavior*
- *Signals*: signals are delivered by the kernel, so they are not guarenteed to arrive at the same execution point
- *Uninitialized memory reads*
- *Behavior on system calls*: it may change over time, e.g. *gettimeofday()*
- *Behavior on shared memory accesses*

The approaches to handle these sources used in PinPlay are:

- PinPlay logs addresses of the memory ranges in use and then preallocates them before replaying
- Code of shared libraries is captured during logging and restored during replay
- Log changes in registers' values after CPU-specific instructions
- Time of signal arrival is logged in terms of instruction count since beginning of execution
- PinPlay skips most of the system calls and restores architecture state after each of them during replay

III. Valgrind overview

Valgrind is an instrumentation framework for building dynamic analysis tools. The program distribution includes six tools, one can also create a new one. Valgrind is licensed under the GNU General Public License, version 2.

The key concept of the framework's architecture is the division between its *core* and *tools*.

The core does low-level work for program instrumentation. It contains: the JIT compiler, low-level memory manager, signal handling unit and a thread scheduler. It also performs system calls for the tool.

The tool is responsible for program instrumentation. Valgrind provides tool programming interface: functions to be called when certain event of interest occurs. The arguments of the functions contain sensible information about the event.

The Valgrind translates code blocks on demand. The translation unit is a block of code ending with one the followings: an instruction limit is reached, a conditional branch is hit, branch to an unknown target is hit. The code translation may be splitted into several phases:

- *Disassembly*: converting machine code into intermediate representation (IR) tree
- *Optimisation 1*: flattening the tree IR, copy and constant propagation, redundant code elimination
- *Instrumentation*: the code block is passed to the tool which transforms it (except the Nullgrind – the tool which does not instrument anything)
- *Optimisation 2*: constant folding and dead code removal
- *Tree building*
- *Instruction selection*: converting tree IR into a list of instructions which use virtual registers
- *Register allocation*: replacing virtual registers with host registers
- *Assembly*: encoding the selected instructions into executable code and writing it to a block of memory

IV. Implementation progress

The key idea of implementation is to use existing tool programming interface. Not only it gives an ability to track events such as system calls and signal arrivals, but also *event-aggregators* such as memory reads (which can be caused by different system calls). This significantly simplifies logging of memory accesses. At the moment regrind (tool for deterministic replay) prototype can work in two modes: *prepare-replay-mode* and *perform-replay-mode*.

In the first mode regrind tracks following events:

- Signal arrival
  1) Before arrival
  2) After arrival
- System calls
  1) Before system call
  2) After system call
- Accesses to memory:
  1) Before memory reads
  2) After memory write

The first two types of events are tracked only for statistic collecting at the moment. But as for the third type of events – accesses to memory – the tool prototype also records accessed memory regions.

Every log point starts with an instruction counter which equals the number of executed guest instructions. "Guest" means that only instructions of program in question are taken into account, not those added by Valgrind. At the moment the log file is written in plain text to simplify debugging.

In the perform-replay-mode regrind restores data about memory accesses. When a memory read occurs regrind gives value from the log file to the tracker functions and thus substitues current value of the memory.

The current implementation does not guarantee exact the same instruction sequence yet but makes visible behavior of some simple programs the same from run to run.

Let's consider an example: linux command *date*.

At first regrind runs in a prepare-replay-mode. The events of interest are memory reads and memory writes. Information about one of them looks like:

```
4e83a:pre_mem_read:tid[1]
:base[4025000]:size[1d]
Fri Mar 30 23:54:50 MSK 2012
```

The first number `4e83a` is an instruction counter. `pre_mem_read` indicates the type of the event. Then arguments of the callback follow: `tid` is thread identifier, `base` is address of memory range and `size` is it's size. Content of the memory region (dump) is presented at the next line of the log file.

In perform-replay-mode regrind executes instrumented guest code. If one of tracked events occurs the tool reads the next checkpoint from the log file, allocates memory and fill it with dump content. Valgrind's core passes arguments to the tool: thread identifier, address and size of the accessed region. Address and size are replaced in the tool with new values.

Thus printed value of the command date will not be equal the actual date and time, but it will be the same as it was at the first run.

Valgrind provides integration with GDB. One can monitor execution of instrumented program. So at the moment it is possible to use both regrind and GDB to perform analysis of reproduced exectuion.

V. Future research

The log file in plain text format is large. It is planned to log data in a binary form to decrease this overhead and provide a simple tool to convert it to a human readable format.

At the moment the regrind prototype does not skip system calls with known results. It is useless work to perform such system calls, so they are to be eliminated in the final version of the tool.

It is also planned to use more information about non-determinism:

- Deliver signals at the same moment as they were delivered at the first run
- Preallocate stack location and used memory regions

Intergation of regrind and GDB allows analyze non-reproducible bugs more efficiently. But integration of regrind with other Valgrind tools will give more powerfull abilities. So it is planned to be implemented in the future too.

VI. Conclusion

The aim of the work is to provide open source tool for deterministic replay of program execution. Existing tools solving the problem are proprietary and not fully available.

The key concept is to use Valgrind – a framework for dynamic binary instrumentation. The framework provides tool programming interface which makes possible to track different events during program execution.

The current implementation reproduces execution of some programs in a seemingly deterministic way (print out does not change from run to run). Some features in the tool are planned to be implemented. The features include covering more sources of non-determinism, improvements of tool architecture and integration with other Valgrind tools.

References

[1] G. Tassey, *The Economic Impacts of Inadequate Infrastructure for Software Testing*, National Institute for Standards Technology, 2002

[2] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallance, V. J. Reddi, K. Hazelwood, *Pin: Building Customized Program Analysis Tools With Dymanic Instrumentation*

[3] D. Bruening, T. Garnett, S. Amarasinghe, *An Infrastructure for Adaptive Dynamic Optimization*, Proceedings of CGO'03, San Francisco, USA, 2003

[4] N. Nethercote, *Dynamic Binary Analysis and Instrumentation*, PhD thesis, University of Cambridge, UK, 2004

[5] N. Nethercote, J. Seward, *Valgrind: A framework for Heavyweight Dynamic Binary Instrumentation*, 2007

[6] http://valgrind.org/

[7] http://pintool.org/

[8] S. Narayanasamy, G. Pokam, B. Calder, *BugNet: Recording Application-Level Execution for Deterministic Replay Debugging*, University of California, USA, 2006

[9] S. Narayanasamy, G. Pokam, B. Calder, *Software Profiling for Deterministic Replay Debugging of User Code*, University of California, USA

[10] H. Patil, C. Pereira, M. Stallcup, G. Lueck, J. Cownie, *PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs*