Pinballs: Portable and Shareable User-level Checkpoints for Reproducible Analysis and Simulation

Harish Patil (Intel Corporation), Trevor E. Carlson (Ghent University)

1. Introduction

Computer architects and researchers use simulation extensively for evaluating various micro-architecture ideas. Most simulators typically take a program binary as input. However, many large, interesting programs require complex execution environments that are hard to setup in conjunction with a simulator. Also, detailed simulation of entire runs of long-running programs can be impractical. To address these concerns, we have developed a user-level check-point format called "Pinball". A pinball is created and consumed using a dynamic instrumentation (Pin [8]) based framework called PinPlay [11].

PinPlay consists of two pin-tools: (i) a *logger* that captures the initial architecture state and non-deterministic events during a program execution in a set of files collectively called a *pinball*; and (ii) a *replayer* that runs on a pinball repeating the captured program execution therein. The replayer can be combined with a simulator allowing simulation based on a pinball instead of a program binary. A pinball can be created for either the entire execution of a program (a whole-program pinball) or for any interesting region of execution (a region pinball). Using region pinballs for simulation can drastically reduce the simulation time for large programs. PinPlay works for both 32-bit and 64-bit x86 program binaries (no source code or special linking required). PinPlay has been ported to multiple operating systems however currently only the Linux version is available externally [3].

The pinball format has many interesting properties:

1. **A pinball is independent of the operating system**. This is because side-effects of system calls are automatically detected [9] and recorded during logging and restored during replay. Only a small subset of system calls (related to thread creation/exit) are executed during replay, the rest are skipped with their side-effects injected from the pinball. At Intel we routinely replay Windows pinballs on Linux to exploit distributed processing for simulation region selection and tracing on a Linux-only pool of machines.
2. **A pinball is self-contained**, i.e. the program binary, input files, special licenses etc. are not needed during replay.
3. **A pinball is relatively small** – since only truly non-reproducible events are recorded in pinballs, typically their sizes are in the range of tens to a couple of hundred MB for most programs with billions of instructions. For example, as reported in [11], for *reference* runs of SPEC CPU2006 [7] programs with average instruction count of 924 billion, the average pinball size is 39MB.

These properties make a pinball an ideal way to share workloads for Pin-based analyses among researchers: once captured a pinball can be analyzed multiple times, anywhere. The use of pinballs for profiling for representative simulation region selection and check-pointing is described in a tutorial [1]. Pinballs are also the basis for reproducible debugging and dynamic slicing of multi-threaded programs described in [14]. We envision two usage models for pinball-based simulation as outlined in section 3. Using the same pinball as input allows an apples-to-apples comparison among simulators. We have made pinballs for the popular SPEC CPU2006 [7] programs available for download (see section 4) which are being actively used by many researchers.

2. The pinball format

The design of the pinball format required a trade-off between generation/consumption runtime overheads and disk space requirement for storage. We chose a design that tries to minimize disk-space requirement by only recording truly non-repeatable events during execution. All other events are recreated during replay. As a result, a pinball is not a *trace* i.e. it is not a sequence of static records but a *checkpoint* that needs to be loaded and *executed* to recreate captured execution. The pinball size minimization came at the cost of added complexity and a large slowdown (typically 100-200X) during logging though. However, the replay is significantly faster than logging (typically $<$ 50X slowdown). This is fine because we expect logging of a pinball to be done only once and replay (with analysis/simulation) multiple times.

Details of various pinball files were omitted from [11] and are now, as implemented in the PinPlay kit [3], described in the Appendix.

3. Using pinballs for simulation

We envision two usage models for pinball-based simulation:

1. *Replayer directly feeding into a Pin-based simulator*: The changes required to make a Pin-based tool to be PinPlay-enabled are minimal. The Sniper simulator [5], which is Pin-based, from Ghent University has been modified, with only a few source line changes, to support simulation of single-threaded pinballs. Use of multi-threaded pinballs for deterministic simulation where shared memory access order from various threads is strictly enforced was reported in [12]. However, many computer architects find enforcing thread order during simulation too restrictive. So we are working on extending multi-threaded pinball-based simulation in the Sniper simulator without enforcing thread order [6].
2. *Replayer-based converter to checkpoints for non-Pin simulators*: At Intel, we convert pinballs to checkpoints in the Long Instruction Trace (LIT) format [13] which most Intel-internal simulators use. The LIT format is very similar to a pinball format consisting of an initial memory image, initial register state, and memory/register injections. A pinball, being Pin-generated, only captures the user-level execution of a program. Whereas a LIT checkpoint, used by simulators simulating a bare metal processor, requires proper processor initialization, a page table etc. During pinball to LIT conversion, we explicitly emit the expected initial values of all system-level registers (some of which are specific to the micro-architecture being simulated). Also, a virtual to physical address mapping is assigned by the converter and a page table is explicitly constructed and added to the initial memory image in the LIT checkpoint. While no external example of this usage model currently exists, we would like to explore collaboration for converting pinballs to checkpoints for QEMU [4] based simulators.

4. Pinballs for SPEC2006

SPEC CPU2006 [7] is a popular benchmark suite used by both computer architects and compiler writers. It is a collection of single-threaded CPU-intensive programs. With the 'reference' inputs, the native run-time of these programs is in the range of few seconds to tens of minutes. However, given the complexity of modern CPUs, detailed cycle-accurate simulation of entire runs of CPU2006 programs could take months! We used the PinPoints methodology [10] to find representative simulation regions for these programs and created pinballs for those regions with the PinPlay logger. Sniper simulator was used to find the quality of representative region selection by comparing the performance results from whole-program simulation with those predicted using simulation of PinPoints pinballs. Both the whole-program and PinPoints pinballs generated are available for download [2]. The average sizes of PinPoint and whole-program pinballs are 12MB and 33MB respectively, which are quite reasonable.

The Sniper simulator is set to simulate PinPoints pinballs and predict whole-program performance. In the last few months, a few dozen Sniper users have successfully downloaded and used the PinPoints pinballs. Some users downloaded the whole-program pinballs instead to generate different PinPoints pinballs by changing certain parameters used for representative region selection.

5. Summary

We have developed tools for generating user-level checkpoints called 'pinballs'. Pinballs are independent of the operating system, are self-contained, and are reasonably small in size. This makes them portable and easily shareable. Simulators can be modified to run with a pinball as input instead or running the original program binary. Because pinballs enable determinstic replay, pinball-based simulation can be completely determinstic resulting in reproducible simulation results. A pinball can be created for either the entire program execution or for only intersting regions of the execution. We have created pinballs (whole-program and region) for SPEC CPU2006 'reference' runs and made them available for download. We call on other reseachers to join the effort of creating a repository of pinballs for other interesting programs which multiple simulator users can share.

6. Acknowledgements

We thank Cristiano Pereira, Jim Cownie, Robert Cohn, Greg Lueck, Mack Stallcup, Brad Calder, Satish Narayanasamy, Ady Tal, Omer Mor, Alex Skaletsky, Lieven Eeckhout, Wim Heirman, and Jeff Reilly for various aspects of this work. Thanks to Geoff Lowney, Moshe Bach, and Peng Tu for supporting this project over the years.

References

[1] HPCA2013 tutorial: Deterministic PinPoints: Representative and repeatable simulation region selection with PinPlay and Sniper. http://snipersim.org/w/Tutorial:HPCA_2013_PinPoints.

[2] Pinballs for SPEC CPU2006 full runs and PinPoints. http://www.snipersim.org/Pinballs.

[3] Program record/replay (PinPlay) toolkit. http://www.pinplay.org.

[4] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46. USENIX, 2005.

[5] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 52:1–52:12, New York, NY, USA, 2011. ACM.

[6] T. E. Carlson, W. Heirman, H. Patil, and L. Eeckhout. Efficient, accurate and reproducible simulation of multi-threaded workloads. In *REPRODUCE: Workshop on Reproducible Research Methodologies*, 2014.

[7] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006.

[8] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI'05*, pages 190–200, 2005.

[9] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *SIGMETRICS'06*, pages 216–227, 2006.

[10] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *MICRO-37*, 2004.

[11] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. CGO'10, pages 2–11.

[12] C. Pereira, H. Patil, and B. Calder. Reproducible simulation of multi-threaded workloads for architecture design exploration. In *IISWC*, pages 173–182, 2008.

[13] R. Singhal, K. Venkatraman, E. Cohn, J. Holm, D. Koufaty, M. Lin, M. Madhav, M. Mattwandel, N. Nidhi, J. Pearce, and M. Seshadri. Performance analysis and validation of the Intel Pentium 4 processor on 90nm technology. In *Intel Technology Journal*, Feb. 2004.

[14] Y. Wang, H. Patil, C. Pereira, G. Lueck, R. Gupta, and I. Neamtiu. DrDebug: Deterministic replay based cyclic debugging with dynamic slicing. In *CGO'14*, 2014.

Appendix

A pinball consists of the following files. Note that this is not an exhaustive list. Some files that are only created when certain debug switches are specified are not listed below. This describes the format ("*log format version: 2.0*") supported by PinPlay kits [3] version 1.1 onwards. The log format of a pinball can be found by searching for '*format*' in a **.result* file.

Files required/used by the replayer

- **.address*: global (common to all threads): various virtual address ranges captured in the pinball.
- **.text*: global: the initial memory image.
- **.result*: per-thread: auxiliary information used by the replayer.
- **.dyn_text*: per-thread: shared library pages tagged by instruction counts.
- **.sel (system effects log)*: per-thread: memory value injections tagged by instruction counts.
- **.reg*: per-thread: multiple register value records for initial register state (*ic* record), registers differing from before and after system calls (*sc* record), context change (*cc* record) etc.
- **.race*: per-thread: records to enforce shared memory access order between threads. e.g. if the file is for thread *i*, the records are of the format '$icount_i$ *j* $icount_j$' implying thread *i* must wait at instruction count $icount_i$ till thread *j* reaches instruction count $icount_j$.
- **.sync_text*: per-thread: records to enforce text page logging and execution order between threads. Its records have the same format as the **.race* file records descibed above.

Files with extra information

- **.log.txt*: global: debug messages during logging.
- **.replay.txt*: global: debug messages during replay.
- **.relog.txt*: global: debug messages during re-logging (replay + logging).
- **.procinfo.xml*: global: information about images/symbols observed during logging.
- **.result_play*: per-thread: auxiliary information generated during replay.