Efficient, Accurate and Reproducible Simulation of Multi-Threaded Workloads

Trevor E. Carlson[1] Wim Heirman[2] Harish Patil[3] Lieven Eeckhout[1]

[1] Ghent University, Belgium
[2] Intel, ExaScience Lab [3] Intel, Software and Services Group

I. Introduction

Architectural design space explorations rely heavily on simulation to quickly compare the performance and energy consumption of large numbers of designs. The simulation infrastructure has a number of important requirements, which at a high level can be described as:

- Efficiency both in time and space, by only simulating relevant parts of the benchmark in detail, avoiding long fast-forwarding or warmup; and maintaining a small disk footprint for storing workloads.
- Accuracy: simulation results must be representative for running the complete workload.
- Reproducible: the unit of work must be fixed across architectures to allow for valid comparisons to be made; workloads must be easily sharable while guaranteeing (mostly) identical simulation results.

Bringing all these goals together in the context of parallel workloads is a real challenge. Design of future many-core processors with large cache capacities require the workload to consist of highly parallel applications with large, realistic input set sizes to both provide adequate scalability and properly exercise the memory subsystem. Additionally, execution-driven simulation, and rather than trace-based simulation, is a necessity to evaluate timing-dependent execution behavior of these multi-threaded applications.

Application sampling is one solution to achieve simulation efficiency, while maintaining an accurate representation of the applications being studied. The ultimate goal is to attain an ease of use by efficiently storing, distributing and replaying workloads, in a reproducible fashion. For single-threaded applications, the PinPoint methodology [6] satisfies all of these requirements. It combines SimPoint [9] to reduce the workload by selecting the most relevant sections, and PinPlay [7] to provide deterministic, reproducible replay. The result is a compact application snapshot starting at the beginning of the SimPoint, and containing all I/O needed to replay the execution exactly. PinPoint removes the need for online application fast-forwarding — which for realistic applications could otherwise result in trillions of executed instructions, contributing a non-negligible amount of time to simulation. Finally, the resulting PinPoint pinball (or PinBall) allows one to replay the exact SimPoint without system library or other dependencies, potentially across platforms [5].

II. Reproducible Parallel Workloads

Unfortunately, parallel application requirements conflict with many of the nice properties that the PinPoint methodology provided for single-threaded workloads. One major issue of multi-threaded applications is that it is much more common for timing behavior, induced by changes in relative thread performance (itself caused by e.g. NUMA effects), to affect the execution path. Timing can, for instance, change the outcome in races for entering critical sections, or can cause dramatic shifts in the work executed by threads when an application uses load balancing or work stealing. Architects will want to see these effects occur in their simulation so they can adapt their hardware or software design accordingly. Fully-deterministic replay is thus no longer desirable as it may impose a thread ordering that is unrealistic for the given target architecture. Instead, timing feedback must be allowed to affect the functional simulation during replay. This is in contrast to PinPlay's default replay behavior, in which all thread orderings are enforced to match those recorded [8].

As application thread alignment can be different on different architectures, taking a snapshot during the execution of an application when running on one architecture does not necessarily provide a valid ordering on the target architecture. One approach, taken in the SimFlex methodology for commercial workloads [10], is to state that any thread ordering is correct as it will probabilistically occur on any architecture. This assumption is valid for request-based parallel applications where threads are independent and rarely interact. In the HPC space, however, threads do interact, and we need to find points in the application that represent *safe* thread orderings, i.e., that are guaranteed to occur in all target architectures. The BarrierPoint methodology [3] exploits the fact that in barrier-based applications (which includes programs based on fork-join parallelism such as OpenMP), whole-program barriers are such safe points for checkpointing by definition. BarrierPoint builds a multi-threaded analogue of SimPoint, where simulation points are no longer delineated by fixed instruction counts, but by whole-program barriers. This makes the BarrierPoint methodology time-efficient and accurate, but its current implementation does not satisfy the reproducibility requirement. In the following section, we will explore what components are missing and how these issues can be solved.

III. Implementation

The original implementation of BarrierPoint used in [3] runs the complete application N times, once for each BarrierPoint. Only the region of interest is simulated in detail, while fast-forwarding is used to proceed past unneeded regions. This simplified the implementation greatly and allowed for parallel speedups by making the simulation of each BarrierPoint independent. However, each BarrierPoint was collected on a different run of the application, potentially even on a different host machine with slightly different OS and libraries. In addition, even after disabling Linux' address space randomization, differences could occur in the memory layout both for shared libraries and stack locations. Therefore, this implementation does not necessarily provide the same virtual address starting point across runs, undermining its reproducibility.

Our goal therefore is to add an additional level of reproducibility without requiring strict determinism throughout the application's execution. We now describe two implementation ideas that would eliminate virtual address differences across runs, while adhering to execution-driven simulation where timing feedback is still allowed to affect the execution path.

1) Linux Checkpoint/Restart: Starting with Linux version 3.11, native checkpoint/restart functionality is available for user-space applications via the CRIU project [1] without requiring additional kernel patches. This is the first step to provide a means for user-space based functional-directed or functional-first simulators to restore the state of an application and restart simulation. Many simulators use Pin [4] as the functional module to drive microarchitectural simulation. By attaching to the process after it has been restored, one can restart application execution and microarchitecture simulation in a deterministic manner. Unfortunately, this solution works only with versions of Linux 3.11 and newer, limiting its current applicability.

2) PinPlay: As an alternative solution, one can use Pin and PinPlay [7] as a means to load an application image and restart execution even on earlier versions of Linux. By converting snapshots captured by CRIU into the PinPlay format, we can now take advantage of application replay capabilities across Linux versions and even allowing the application to run on other operating systems.

Although we describe user-space simulation solutions here, full-system snapshot capabilities have existed for some time and allows for BarrierPoint to be implemented on full-system simulation platforms.

Limitations

Physical addresses can pose a simulation problem for reproducibility with user-space simulation. Sniper [2] supports automatic extraction of virtual-to-physical mappings as introduced by the operating system to allow one to model data sharing between processes. But these mappings will not be restored in exactly the same way in C/R or PinPlay as the exact physical page mapping location is not typically tracked, or would be difficult to re-establish after application restoration. A solution to this issue is to maintain virtual-to-physical mappings that have been used and to restore those mappings to the simulator at restore time.

The virtual to physical mappings used by the operating system are normally not saved for Linux's checkpoint-restart system. Therefore, this data will need to be recorded and injected into the simulator to maintain an accurate system view.

IV. Conclusion

We provide a number of potential solutions that expands the applicability of user-space sampling methodologies by providing determinism for the BarrierPoint methodology (and potentially other sampling methodologies). With a two-pronged approach though the use of reproducible program state, while still allowing non-determinism for architectural exploration and evaluation, we feel that this methodology holds promise by reducing the number of simulations required to obtain high statistical confidence in the simulated results.

References

[1] Checkpoint/restore in userspace (CRIU). http://criu.org.

[2] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011.

[3] T. E. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout, "BarrierPoint: Sampled simulation of multi-threaded applications," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, to appear.

[4] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, Jun. 2005, pp. 190–200.

[5] H. Patil and T. E. Carlson, "Pinballs: Portable and shareable user-level checkpoints for reproducible analysis and simulation," in *REPRODUCE: Workshop on Reproducible Research Methodologies*, 2014, submitted for review.

[6] 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 *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2004, pp. 81–92.

[7] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Apr. 2010, pp. 2–11.

[8] C. Pereira, H. Patil, and B. Calder, "Reproducible simulation of multi-threaded workloads for architecture design exploration," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, Sep. 2008, pp. 173–182.

[9] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2002, pp. 45–57.

[10] T. Wenisch, R. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. Hoe, "SimFlex: Statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, no. 4, pp. 18–31, Jul./Aug. 2006.