Accurate Functional-First Multicore Simulators

Derek Chiou, Hari Angepat, Nikhil A. Patil, and Dam Sunwoo
Department of Electrical and Computer Engineering
University of Texas at Austin
{derek,angepat,npatil,sunwoo}@ece.utexas.edu

Abstract—Fast and accurate simulation of multicore systems requires a parallelized simulator. This paper describes a novel method to build parallelizable and cycle-accurate-capable functional-first simulators of multicore targets.

1 Introduction

Accurate simulators used at Intel [8] and AMD [2] simulate single core targets[1] on single core hosts at 1KIPS to 10KIPS, requiring somewhere between one year to ten years to simulate a 3GHz target for 2 minutes. To make matters worse, simulators are getting slower as the number of target cores and their complexity increase. Parallelization is needed to improve simulator performance.

Our FPGA-Accelerated Simulation Technologies (FAST) project [6] is researching techniques to parallelize simulators. FAST simulators are factored into a functional model (FM), that models the target functionality (ISA and peripherals), and a timing model (TM), that models the target micro-architecture. The FM executes instructions independently of the TM and passes a FM-execution order instruction trace to the TM that uses the trace to simulate those instructions in the micro-architecture.

Target timing affects which instructions are fetched and the order they are executed, potentially resulting in the FM fetch/execute path[2] diverging from the target. In a FAST simulator, when the TM detects divergences, it guides the FM back to the target path. For example, a branch misprediction causes the target to fetch wrong path instructions while the FM may not, causing fetch path divergence. Upon detection, the TM guides the FM to roll back to the mispredicted branch and take the wrong path. When the branch is resolved by the target, the TM guides the FM to roll back and return to the right path.

FAST is a *simulator-level speculative functional-first* simulator since the FM speculatively executes instructions and allows the TM to correct it when necessary. Some functional-first simulators like FastSim [15] execute in target order by frequent querying of the TM. For example, its FM queries the TM branch predictor at every branch. Doing so eliminates simulator misspeculation but virtually eliminates the possibility of running the FM and TM in parallel.

Our previous work [4], [5], [6], [7], [16] introduced simulator-level speculation and its ability to drastically reduce FM/TM coupling, making parallelization at the FM/TM boundary efficient and thus enabling the FM and TM to be implemented in different technologies. Our initial FAST prototype implements the FM in a modified software virtual machine (QEMU [16]) running on commercial microprocessors and the TM in an FPGA.[3] It is capable of cycle-level accuracy while simulating single core x86 targets running Windows and Linux at 1MIPS-10MIPS, but does not yet support multicore targets. Methodologies to simplify TM development while maintaining accuracy, flexibility, and performance have been developed [7].

The main contribution of this paper is the *target memory oracle (TMO)* that enables functional-first simulators and their derivatives to efficiently and accurately detect and correct execution path divergences, especially for memory operations, a critical problem for multicore targets. Our solution maintains the advantages of functional-first simulators, including high simulation speeds and simple, reusable FMs. It also gracefully supports the use of a parallelized FM, providing scalable, accurate simulation of parallel computer systems.

2 The Problem

Execution path divergence is common for out-of-order targets since there is more than one legal execution order. However, as long as the target obeys dependencies, its functionality will be equivalent to any other implementation that also obeys dependencies. Thus, given a functionally-correct FM, most execution path divergence can be safely ignored since it does not affect functional target-correctness.

Unfortunately, some dependencies depend on timing and thus cannot be determined statically. Two operations on shared state (memory[4]) executed by separate entities generally do not have a well defined execution order even though the two operations may be dependent for certain execution orders. Though memory consistency models restrict what orders of reads and writes are legal, there are often multiple legal orders. Thus, it is likely that the FM sometimes executes reads/writes in a different order than the target. We call such divergence in the functional/target state-access order the *reordering problem*, a long-standing issue that has caused some to abandon functional-first simulation for multicore[5] targets [3].

Though reorderings occur in primarily single core targets due to entities like I/O devices that access shared memory, they

Manuscript submitted: 05-May-2009. Manuscript accepted: 14-Jul-2009. Final manuscript received: 21-Jul-2009.

1. We use the term *target* to mean the machine being simulated and the term *host* to mean the platform on which the simulator runs.

2. We call the target fetched/executed instruction order, including both right path and wrong path instructions, the *target fetch/execute path* and the FM fetched/executed instruction order, the *functional fetch/execute path*.

3. Of course, the TM can also be implemented in software or in a software simulation of a TM implemented in an FPGA.

4. In the context of reordering we use the term memory to mean any state that could be reordered, including shared memory, machine state registers, cycle registers, and I/O device registers.

5. We use the term multicore to mean multiple entities (cores, I/O devices, etc.) that can read/write to the same state. Thus, even single core systems with I/O devices have multicore issues.

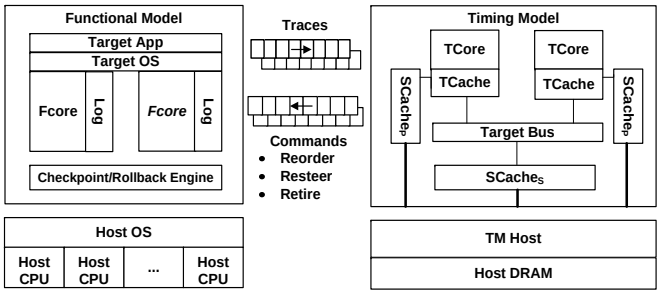

Fig. 1. FAST Simulator of a Parallel Computer System

are infrequent and rarely affect functionality. In a multicore target, however, reorderings can have a significant effect on functionality and performance. In well-labeled shared memory programs, reorderings only occur at lock variables but reordering locks results in a very different computation. Not-well-labeled programs exist for a variety of reasons — intentionally introduced non-determinism, transactional memory, an unintended error — making any access to a memory location potentially reordered. If allowed to propagate, reorderings will likely result in functional and/or timing errors. A common solution is a *timing-directed simulator* where the TM tells the FM when and what to fetch, execute, etc. In such schemes parallelism between the FM and TM is limited meaning the FM and TM must be tightly integrated for performance and complicates the FM since it must support micro-architectural structures like rename, reservation stations, etc.

3 Parallelized Multicore Simulators

A simplified FAST simulator of a multicore target is shown in Figure 1. The FM consists of multiple Fcores, each simulating a target component such as a core, an I/O device, etc. The FM is assumed to be parallelized across multiple host cores, with a potentially different memory model than the target, for higher simulation performance. Conceptually, each Fcore has a corresponding Tcore in the TM. Each Fcore generates its own trace that is passed to its partner Tcore. If the Tcore determines that the trace does not match the target core, it guides the Fcore to correct the trace.

Reorderings would be conceptually trivial to detect (but not correct) if a combined trace, containing the global order of all of the memory operations from all of the target cores, were available[6]. There is, however, no support to determine memory operation execution order on a conventional parallel host, making it difficult use such hosts. Researchers are studying hardware support to address this omission [1], [10], [13], [17], but no such solution is commercially available. Lamport clocks [11] can generate a global order of events on conventional hardware but they require correct timestamps. Because memory accesses can be reordered, generating the correct timestamp value would likely require an expensive memory barrier between each memory operation to be ordered. Also, even though reordering detection is straightforward with a combined trace, correction is not, requiring extracting the target execution order from the TM and rolling back and re-executing in that order on the FM.

6. A global order is impossible for some target architectures. For example, the accesses in a memory model that permits reading a non-local value before it is globally visible may not be expressible in a single global order.

4 Our Solution

Our solution is based on indistinguishability — as long as all data values returned to functional loads are the same as those returned to the corresponding target loads, the functional model is target-correct. We also assume that it is trivial for the functional model to know what values it loads and stores.

Loads[7] are the only way for a reordering to be observed and its effects felt by the FM. All other instructions are assumed to obey program dependencies when accessing state and thus cannot result in a reordering. To ensure functional load values are consistent with target load values, the TM models the target memory system *including the data values* in the target memory oracle (TMO). When the TM "executes" a load[8], the target load value at that precise target time is read from the TMO and compared to the functional load value that was passed to the TM as part of the trace. If the values are the same, either there was not a reordering or the functional value was the same as the target value, resulting in target-correct functional behavior. **If the values are different, the TM immediately freezes, ensuring that it does not get corrupted by using incorrect data from the FM, and provides the target value and the corresponding dynamic instruction number of the load to the FM.** The FM regenerates the trace to reflect the corrected functional load value (Section 5.) This technique naturally eliminates any reordering that does not affect functionality, minimizing rollbacks.

How is the TMO updated and with what data? The TM will not execute a store until all instructions it depends on, including reordered loads, have been executed target-correctly by the FM. If it executes before, it may have an incorrect address and/or data due to loads that it depends on not yet having been corrected. **Thus, the functional store value itself is target-correct at the time the TM executes that store and can be used to update the TMO.**

By maintaining target data in the appropriate TM component models (store buffers, caches, etc.) and passing the functional load values to the TM, the functional/target value comparison can be done at the right target time. The state of the TM component models itself can be cached at the *simulator-level*, reducing the modeling cost of large target structures. Because each simulator-level cache (Scache$_p$) caches a private namespace (target cache model, store buffer model, etc.), it is a private cache not requiring simulator-level coherence. As TM components simulate target coherence, however, the simulator-level caches will automatically be kept coherent. For example, if the target cache (Tcache) snoops an invalidation of address X, it first accesses its simulator cache (Scache$_p$). If its Scache misses, the Tcache freezes while the Scache fetches the required data that is used to determine whether the Tcache hit or missed. Since the TM is using the same basic structures as the target, its performance scales as the target's performance scales.

It may be desirable, however, to not have to include data in every target structure model. When memory accesses can be represented by a global order (not necessarily program order) of memory operations, TMO data can mostly reside in a single, shared memory that is accessed at the correct target time. Memory consistency models such as the IBM 370, Sun

7. We use the terms load/store to refer to accesses to memory, including accesses originating from instructions that may not be a load, such as a register-memory ADD instruction.

8. Though similar to value prediction, this approach does not break consistency models [12] because checks are done at execute rather than commit.

TSO and PSO, Weak Ordering, and Alpha do not permit a write from another core to be read until that write is globally visible and thus have a global order. Implementing a single shared memory is straightforward for small targets, due to the relative speed of the simulator compared to host memory (10MIPS simulation speed is 100ns per instruction.) Simulator-level caches ($Scache_s$) could be introduced in front of the single TMO to reduce effective latency, but require simulator-level coherence if there is more than one.

Functional memory does not need to be kept consistent with the target memory since any discrepancies will be detected as a reordering and corrected. Therefore, the FM can run independently of the TM, thereby increasing parallelism. Every handled reordering, however, degrades performance, making it useful to keep functional memory as close as possible to target memory.

It may appear that the TMO requires separate memory from the functional memory, effectively doubling the memory requirements. However, the functional memory can be implemented as a cache of the TMO or vice-versa. An efficient way to implement the former is to copy the corresponding TMO page to a functional page, *overwriting the original contents*, on every functional TLB miss and fill. One can use as many or as few functional pages as desired. However, statically allocating one functional page to each functional TLB entry eliminates the need for functional page management. If the page is read-only, a copy is not necessary; the functional TLB could simply be made to point at the corresponding TMO page.

Reorderings only occur when the values loaded from shared state actually differ between a likely-in-order FM and the target, which is rare in high performance targets (e.g., locks are taken infrequently.) Thus, we expect TMO-based simulators to perform well in most cases.

Note that the oracle technique is general and allows the TM to correct anything that the FM speculated incorrectly. For example, we have been using the same technique for branch prediction and interrupts even in our unicore simulators.

Limited space only allows comparison to a subset of related work. The described technique is similar to parallel replay that provides architectural support to record and deterministically replay parallel execution [10], [13], [14], [17]. These techniques, however, generally record addresses, rather than values, hampering correction. Our work is also similar to work on checkpointed processors, especially silent deterministic replay that uses previously loaded values to avoid read-after-write issues [9]. The TMO was, however, invented independently to accelerate simulation. Though replaying with logged values is the same, the TMO both checks for divergence and provides the data to correct.

5 Functional Model Support

The functional model must support (i) rollback to a past, target uncommitted instruction, (ii) correction of either the reordered load value or a different branch direction and (iii) continuing from that correction reusing all previous corrections to uncommitted instructions. To do so, the FM operates in two modes: execute and replay. During an execute phase, the FM executes as it would naturally, loading from and storing to functional memory while generating each Fcore's trace and sending it to the TM. Each dynamic branch target is logged in a branch target log and each dynamic load value is logged in a load log.

During a replay phase, (i) instructions are processed in the same order as the previous execute/replay by using the previous branch target log rather than re-executing branches, (ii) loads are provided with a logged load value, rather than re-executing the load, (iii) stores are re-executed, including writing to functional memory, to reduce functional/target memory divergence, (iv) all other instructions are re-executed, and (v) trace entries are updated with new addresses for loads and new store values. Replay is distinct from execute because every load uses the load log value and every branch uses the branch target log, enabling multiple corrections.

There are precise transition points between phases. Normally, the FM is in execute mode. If the FM fetch path diverges, such as in the case of a branch misprediction, a branch resolution of a misprediction, or an interrupt or exception, the TM notifies the FM of the instruction that diverges and how it diverges. In response, the FM rolls back to the last common instruction, modifies the branch target log to reflect the new destination, and then *executes* from that point.

When the TM detects a reordered load, it rolls back the FM to that load and corrects it. It may appear that the FM must then *re-execute*, rather then replay, instructions dependent on the reordered load. Though re-executing dependent instructions works, determining which instructions are dependent is cumbersome. Instead, one can simply correct the load log entry for the reordered instruction and then replay from that instruction. Instructions already executed by the TM should replay identically to the previous execution, meaning the updated trace entries should be identical to the overwritten values, providing either an opportunity for additional verification (compare the original with the updated value) or an opportunity to avoid work (do not update the trace entry.) For instructions that have not yet been executed by the TM, however, addresses and data may change. Recall that the TM freezes immediately after it detects the reordering and restarts only after the trace entries for un-executed instructions have been updated (the TM can check instruction-by-instruction, allowing it to restart before all of the trace entries have been updated.) By the time the TM actually executes an instruction, it has target-correct information, including the correct address for a load and the correct address and data for a store. Even if a functional load whose address changed is replayed with the original data value read from the incorrect address, the TM can detect it as a reordering and handle it accordingly.

6 Example

We demonstrate an example of multiple rollbacks over the same code to correct multiple divergences from the target. Figure 2 shows pointer-chasing code using three memory locations and their initial values. Consider the case where the functional order is A,B,C,D,J while the target order is D,A,B,C,E. When the TM executes A, it discovers that A is reordered by comparing the functional and target load values. The TM freezes and requests that the FM rolls back P2 to the instruction immediately prior to A, injects the correct load value for A into the log, then replays the instructions starting from A while regenerating the trace. B is replayed with the wrong load value, but the right load address through R0, allowing B to be correctly checked for reordering by the TM. As subsequent functional loads are checked and found to be reordered, they are corrected and replayed in the same way, preserving all previous corrections.

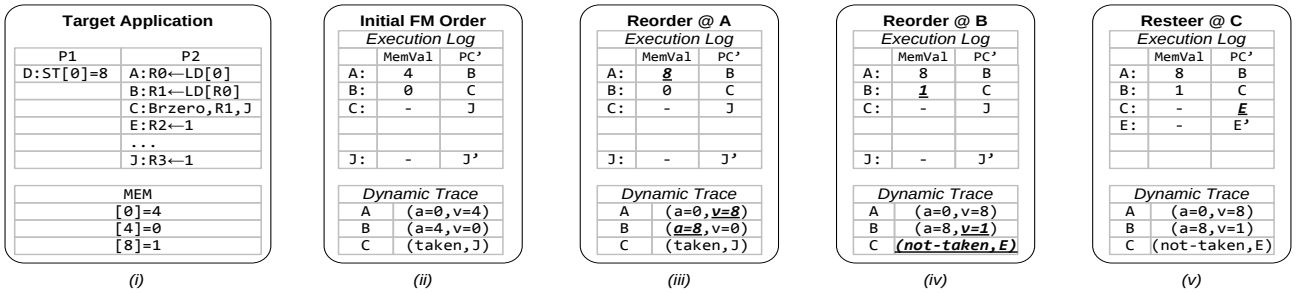

Fig. 2. Handling Multiple Reorders. (i) depicts a simple pointer-chasing code snippet with the initial memory contents as shown. (ii) to (iv) depict the functional log (with associated load values (MemVal) and next PC (PC')) and dynamic trace (address (a), load values (v), branch targets) during incremental TMO reordering handling on processor P2. (v) depicts the final resteer necessary to follow the target branch mispredict at C. Fields that change between iterations are underlined and in bold.

The TM can compare the predicted instruction addresses with the functional instruction addresses to determine if the functional fetch path needs to be corrected. Such corrections are independent of any reordering of loads since they are only a function of the branch predictor. In our example, the target predicts instruction C as taken, the initial functional and target fetch paths are identical so no correction is necessary. Sometime after correcting the reorder at B and changing the value of R1 to 1, the TM executes branch C and discovers target misspeculation. Thus, complete target accuracy is preserved in the presence of both multiple dependent memory reorders as well as overlapping control path speculation.

It is important to note that while multiple reorderings are supported we expect this to be infrequent for scalable target applications and architectures. As a result, we optimize for the common-case where functional execution order is indistinguishable from target order.

7 Conclusions and Future Work

The TMO enables a functional-first simulator to dynamically detect and correct functionality when required to accurately model a target. Thus, the TMO allows an FM to speculatively execute ahead of the TM, even for a parallel target running on a parallel host, making the common case very fast. The TMO enables an in-order FM to model any arbitrary target at the cost of corrections.

We are currently in the process of implementing the described methodology and expect to have a working prototype soon. It appears that logging values and branches has very little impact on the performance of our FM. We are also parallelizing our FM to run on a parallel host. We expect roughly 10MIPS of scalable cycle-accurate-capable performance per host core, roughly three to four orders of magnitude faster than industry simulators per host core.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants No. 0615352 and No. 0747438 and gifts from Intel and IBM. We thank the anonymous reviewers for their comments.

References

[1] D. F. Bacon and S. C. Goldstein, "Hardware-assisted replay of multiprocessor programs," in *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, 1991.

[2] R. Bhargava, L. Barnes, and B. Sander, "AMD," personal email communication, Aug. 2007.

[3] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt, "Network-Oriented Full-System Simulation using M5," in *Sixth Workshop on Computer Architecture Evaluation using Commerical Workloads*, Feb. 2003.

[4] D. Chiou, "FAST: FPGA-based Acceleration of Simulator Timing models," in *Proceedings of the first Workshop on Architecture Research using FPGA Platforms, held in conjunction with HPCA-11, San Francisco, CA*, Feb. 2005.

[5] D. Chiou, H. Sanjeliwala, D. Sunwoo, J. Xu, and N. Patil, "FPGA-based Fast, Cycle-Accurate, Full-System Simulators," in *Proceedings of the second Workshop on Architecture Research using FPGA Platforms, held in conjunction with HPCA-12, Austin, TX*, Feb. 2006.

[6] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. H. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators," in *Proceedings of MICRO*, Dec. 2007, pp. 249–261.

[7] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. H. Reinhart, D. E. Johnson, and Z. Xu, "The FAST Methodology for High-Speed SoC/Computer Simulation," in *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, Nov. 2007, pp. 295–302.

[8] J. Emer, "Intel," personal communication, Jan. 2008.

[9] A. Hilton and A. Roth, "Decoupled Store Completion/Silent Deterministic Replay: Enabling Scalable Data Memory for CPR/CFP Processors," in *ISCA*, Jun. 2009.

[10] D. R. Hower and M. D. Hill, "Rerun: Exploiting Episodes for Lightweight Memory Race Recording," in *ISCA*, Jun. 2008.

[11] L. Lamport, "How to make a multiprocessor that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. 28, no. 9, pp. 690–691, Sep. 1979.

[12] M. M. K. Martin, D. J. Sorin, H. W. Cain, M. D. Hill, and M. H. Lipasti, "Correctly Implementing Value Prediction in Microprocessors that Support Multithreading or Multiprocessing," in *MICRO*, Dec. 2001.

[13] P. Montesinos, L. Ceze, and J. Torrellas, "Delorean: Recording and deterministically replaying shared-memory multiprocessor execution effciently," in *ISCA*, 2008, pp. 289–300.

[14] S. Narayanasamy, G. Pokam, and B. Calder, "Bugnet: Continuously recording program execution for deterministic replay debugging," in *ISCA*, 2005, pp. 284–295.

[15] E. Schnarr and J. R. Larus, "Fast out-of-order processor simulation using memoization," in *Proceedings of ASPLOS*, Oct. 1998, pp. 283–294.

[16] D. Sunwoo, J. Kim, and D. Chiou, "QUICK: A Flexible Full-System Functional Model," in *Proceedings of ISPASS*, Apr. 2009, pp. 249–258.

[17] M. Xu, R. Bodik, and M. D. Hill, "A "flight data recorder" for enabling full-system multiprocessor deterministic replay," in *ISCA*, 2003, pp. 122–135.