Rapid Development of Error-Free Architectural Simulators using Dynamic Runtime Testing

Saša Tomić Adrián Cristal Osman Unsal Mateo Valero
Barcelona Supercomputing Center, Spain
{firstname.lastname}@bsc.es

Abstract

Architectural simulator platforms are particularly complex and error-prone programs that aim to simulate all hardware details of a given target architecture. The development of a stable cycle-accurate architectural simulator can easily take several man-years. Discovering and fixing all visible errors in the simulator often requires significant effort, much higher than for writing the simulator in the first place. In addition, there are no guarantees that all programming errors will be eliminated, no matter how much effort is put into it.

This paper presents dynamic runtime testing, a methodology for rapid development and accurate error detection in architectural cycle-accurate simulators. In dynamic runtime testing, the simulator execution is dynamically compared with a simple and functionally equivalent emulator. A possible error is detected if any instruction produces different results in the simulator and the emulator.

Dynamic testing can help the developers of architectural simulators to get a reliable and accurate verification of functional correctness. Based on our experience, dynamic testing reduced the simulator modification time from 12-18 person-months to 3-4 person-months, and it only modestly reduced the simulator performance (in our case under 20%).

1. Introduction

The proposals for hardware changes are typically first implemented and evaluated on architectural cycle-accurate simulators. These simulators aim to accurately represent the functionality, the interaction, and the timing of all functional blocks of the real hardware. As such, architectural simulators are typically very complex and error-prone. A non-working simulator can unnecessarily delay the evaluations of architectural proposals. Incorrect simulator evaluations can take future product development in a wrong direction, or create other unnecessary development costs. Simulator developers often invest significant effort in thoroughly testing and verifying the simulators, attempting to confront the errors.

It is commonly estimated by many hardware and software companies that verification will take between 50 and 70 percent of the total product cost [6, 8]. For large or mission-critical projects, verification can take as much as 90 percent of the total cost. Verification and debugging are often seen as the most difficult problems in today's complex hardware and software systems. This is especially the case with the products that require continual modifications both before and after being released. Traditional testing methods (for example, unit testing [11]) require a significant amount of programming effort to provide good confidence in simulator correctness. However, architectural simulators are often changed rapidly and extensively, used to evaluate a certain idea or approach, and after that the changes are discarded. Thus, testing of architectural simulators is often performed irregularly and unsystematically.

In contrast with simulators, architectural emulators (for example, QEMU [1]) are not concerned with the details of hardware architecture. The functionality of the emulators typically consists of: (1) decoding instructions, (2) executing them, and (3) updating the simulated memory. The objective of the emulators is to provide the functional equivalent of the target architecture and not to provide the performance estimates for the target architecture. Emulators are typically used to make virtual machines and to do cross-platform software development. Since emulators are much simpler than the simulators, they are generally easier to debug and validate than simulators and, therefore, much more stable. Still, executions on an architectural simulator and an emulator have to produce identical final results.

In this paper, we present a technique for discovering the unintentional functional errors, or bugs, in the architectural cycle-accurate simulators. We propose a *dynamic runtime testing* methodology, which leverages the execution of an emulator to verify the functional correctness of the architectural simulator. The key idea of dynamic testing is to

execute the simulator and the emulator side-by-side, and to compare their execution as often as possible, preferably after every instruction execution. Any difference in their execution indicates a possible bug in the simulator and needs to be carefully examined. The functional correctness of the simulator is tested dynamically during entire simulator execution, in every simulator execution, and during entire simulator lifetime. In Section 2, we explain how dynamic testing can be applied to practically any architectural simulator, either to an entire simulator, or to a specific functional block of the simulator. Then we show several use cases of the methodology: hardware transactional memory, coherent multi-level caches, and out-of-order processors.

After dynamic testing reports a potential bug, a simulator developer has enough data to find it and fix it, quickly and efficiently. In Section 3, we explain our preferred debugging methods – execution tracing and an interactive debugger tool.

In Section 4, we evaluate the performance impact of dynamic runtime testing, applied to the cycle-accurate simulators of coherent multi-level caches and Hardware Transactional Memory (HTM). The overhead of dynamic testing is modest (10-20%) in our implementations, since the base simulators of caches and HTMs are much more complex than their functionally equivalent emulators. The overhead of dynamic testing could be even smaller than in our implementations, if a highly complex base simulator (e.g., a pipelined out-of-order architectural processor simulator) is dynamically tested with a well-optimized architectural emulator. While fast cycle-accurate architectural simulators can simulate around 2 MIPS (million instructions per second), their functionally equivalent emulators can even provide a near-native execution speed [1]. Even if we assume a 10 times slowdown in an emulator, on a modern machine this is more than 1000 MIPS, which is 500 times faster than the complex base simulator. Thus, the overhead of such a configuration could be less than 1%.

In Section 5, we share our experiences with using dynamic testing. Dynamic testing helped us to rapid develop, testing, and verify several architectural cycle-accurate simulators. Consequently, our simulator development became more productive and more efficient. In particular, dynamic testing provided us the following advantages over other simulator testing methods:

1. Faster simulator testing, since we did not need to create a complex and extensive test suite,
2. Faster simulator debugging, since we could pinpoint a precise moment and the circumstances that lead to a bug, instead of only discovering that a bug appeared, and
3. Faster simulator development, since we had more confidence and freedom to develop the simulator, knowing that any introduced bug will immediately appear.

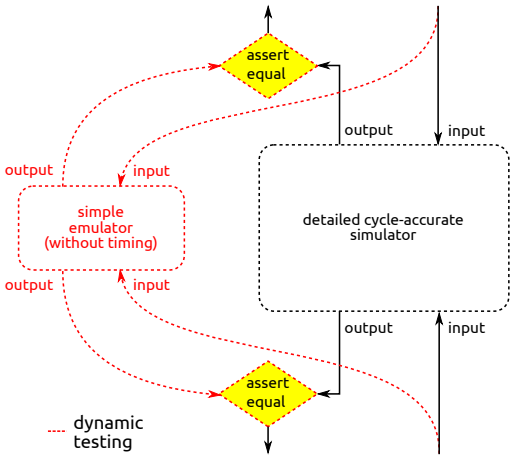

Figure 1. An overview of dynamic testing. The tested simulator (black) and the functionally identical emulator (red) have to be produced the same output during entire simulator execution. Any difference indicates a likely bug.

In addition, the method could help to recover the simulator from a certain type of bugs, by falling back to the execution results of the emulator. This can improve the overall reliability of the simulator (although not its correctness).

Dynamic runtime testing is related to the classical back-to-back software-testing methodology [14]. Classical back-to-back testing consists of comparing the execution of two independent and functionally equivalent programs. The programs are compared: (1) statically (for example, by reading the source code), (2) with specially designed functional test cases, and (3) with random test cases. However, in back-to-back testing methodology the developers need to dedicate significant time to creating a large collection of test cases. In contrast, dynamic runtime testing is a small and one-time development effort.

2. Detecting Bugs Using Dynamic Testing

Conventional debugging methods help discover how and why a bug occurred, but offer very little help to discover *whether* and *where* a bug occurred. Assertions [7] in the program may detect some bugs. However, not only they pollute the source code and require a significant effort from the developer, but they also catch only the execution cases which the developer believes are definitely incorrect. In addition, an assertion may fail millions of cycles after a bug occurred, and the bug could have propagated through many

intermediate values.

If an assertion failed, or the execution gives an incorrect result, the developer becomes aware of a bug. However, he could start looking for the bug anywhere in the recent execution. To find the bug fast, a developer has to rely on his intuition and on his previous programming and debugging experience. Therefore, finding bugs is often seen more of an art than a science.

The simulators are used for developing and evaluating novel hardware designs, and this adds to the simulators an additional source of bugs. A flaw in the hardware design can cause incorrect simulator functionality, similarly to an ordinary software bug. However, subtle hardware design bugs are often uncovered only in a detailed hardware simulation.

A particular complexity of finding bugs is that many bugs appear benign at the moment they happen (for example, a value is 2 instead of 3), and may (or may not) affect the execution millions of instructions later. A wrong variable value may affect the execution in different ways. The execution may: (1) fail, (2) terminate with the incorrect result, or (3) terminate with the correct result. The bugs that do not manifest, that is, where executions still correctly terminate, are the most difficult to detect. This kind of bugs is equally important, since wrong values might make an execution shorter or longer than it should be (for example, wrong number of loop iterations).

Dynamic testing (illustrated in Figure 1) can help discover whether and where a functional bug in the simulator occurred. It can be applied both to the entire simulator (example in Section 2.4), or only a certain functional block of the simulator (examples in Sections 2.1 and 2.2). Dynamic testing consists of comparing the outputs of a functional block of a simulator, with the outputs of a functionally equivalent emulator of the same block. The comparison is done with every executed operation, and all outputs have to be identical. Although any type of output from a block can be compared, we were comparing the values of the memory accesses.

The procedure for implementing dynamically testing is the following. **Emulator integration.** We make and integrate a functionally-equivalent emulator of the tested functional block. The block emulator should not provide any timing estimations, and instead it should focus on being simple, well performing, and functionally correct. **Emulator validation.** We disable the block simulator and redirect its input (e.g., operations and memory values) to the block emulator. All applications should terminate correctly with the emulator, without giving any errors or warnings. **Simulator-emulator comparison.** Finally, we re-enable the block simulator and share its input with the block emulator. In our implementations, we sequentially execute each operation with the block simulator and then on the block emulator, giving them the same input. We were not executing the simulator and the emulator in parallel (multithreaded), although it can be done. Parallel execution would unnecessarily complicate the code in our case and would not significantly change the simulator performance. If there is any difference in the output from the dynamically tested functional block of the simulator and the emulator (e.g., they write different values to memory), the simulator has to notify the developer and provide an exact state when the difference appeared. If there is no difference, a developer has high confidence that his simulator-based evaluations were functionally correct. In this case, any potentially remaining bugs in the simulator did not manifest in the executions.

It is possible that some bugs remain in the emulator after the emulator-validation phase. However, the bugs in the simulator and the emulator will most likely produce different outputs at some point of the executions, in the later phase of simulator-emulator comparison. These uncommon bugs can be eliminated in this phase.

In the following sections, we demonstrate dynamic testing with several real-world use cases.

2.1. Use Case: Hardware Transactional Memory

In our past work, our group implemented and evaluated several proposals for Hardware Transactional Memory (HTM). Transactional Memory [5] is an optimistic concurrency mechanism, which allows different threads to execute speculatively the same critical section, in a "transaction". In case a speculation is successful, the transaction "commits". Otherwise, we say that there is a conflict between transactions, and the transaction is aborted – that is, the speculative changes are rolled back and the transaction is re-executed.

The most bug-prone part in our HTM implementations was the clearing of speculative states for cache lines, during transaction commit and abort procedures. For better performance or efficiency, an HTM implementation may partially clean the speculative states [10], or to group-change the permissions of all speculatively modified lines [4]. Our simple HTM emulator has to avoid the complexity of these proposals and to provide a correct, reference HTM implementation. Figure 2 illustrates one solution. We store speculatively modified cache lines in a map from the C++ Standard Template Library (STL), and the speculative reads in an STL set. The STL map associates a cache line address with a copy of cache line data. The STL map and STL set have practically unlimited capacity, and this further simplifies the HTM emulator. All HTM emulator operations are performed instantly and reliably.

The presented simple HTM emulator can be used to test HTMs with both eager and lazy version management, since all HTMs satisfy the following two invariants. First, all reads have to return either: (1) the last value speculatively

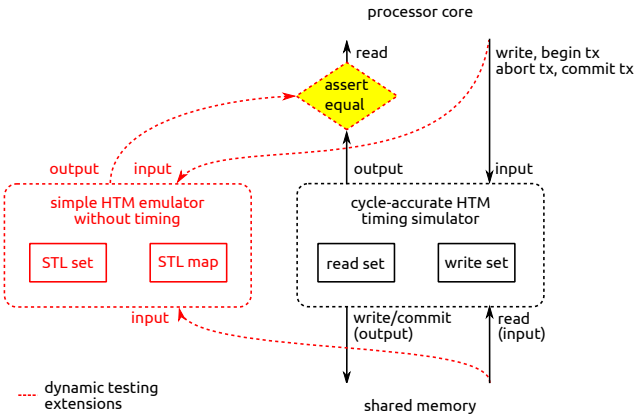

Figure 2. Dynamic runtime testing applied to HTMs. All reads are compared between the HTM simulator and the HTM emulator, and must return the same value. Optionally, writes/commits could be compared as well.

TRANSACTIONAL READ

```
data             =          HTM.write_set[txid].get(addr) or caches.get(addr)
data_functional = FunctionalHTM.write_set[txid].get(addr) or shr_mem.get(addr)
if (data_functional != data) {
   FATAL_ERROR("incorrect HTM value: %x instead of %x", data, data_functional);
   // also print the simulator cycle, state, and the accessed address, and then exit
}
```

TRANSACTIONAL WRITE

```
if not HTM.write_set[txid].has(addr):
   HTM.write_set[txid].fetch_from_caches(addr)
HTM.write_set[txid].update(addr) with data

if not FunctionalHTM.write_set[txid].has(addr):
   FunctionalHTM.write_set[txid].fetch_from_shr_mem(addr)
FunctionalHTM.write_set[txid].update(addr) with data
```

ABORT

```
FunctionalHTM.abort_instantly(txid)  // instantly clears the write_set & restarts
HTM.initiate_abort(txid)             // rollback & restart; may take many cycles
```

COMMIT

```
FunctionalHTM.abort_conflicting_transactions(txid) // detect & resolve conflicts
FunctionalHTM.commit_to_shr_mem(txid)  // instantly publishes specul. changes

// regular HTM: starts conflict detection/resolution/committing specul. changes
// this may take many cycles
HTM.initiate_commit(txid)
```

Figure 3. Pseudocode of the actual implementation of dynamic testing for HTM

written in the current transaction, or (2) the non-speculative value speculatively written outside of a transaction. Second, the non-speculative writes to the shared memory have to be equal to either: (1) a speculative value that a transaction commits, or (2) a non-speculative value written outside of a transaction. It was sufficient for us to test only the values of memory reads. Figure 3 shows the pseudo-code of our implementation of dynamic testing for HTMs.

During the execution, a simulated processor core sends (1) the memory accesses and (2) the transactional events, to both the HTM simulator and the HTM emulator. The value of every read is compared between the two HTMs, and any difference indicates a likely bug in HTM simulator, and is logged together with more details on the simulator state (for example, simulator clock). Based on these logs, the developer can debug the simulator, knowing the location and the time the potential bug appeared. After eliminating the bug, he needs to repeat the evaluations and check for remaining or newly created bugs. The process is repeated until dynamic testing reports an execution without any difference between the HTM simulator and the HTM emulator. At this point, any potentially remaining simulator bugs do not impact the simulator functionality. The timing estimation of the simulator may still be incorrect, since dynamic testing does not guarantee the detection of timing estimation bugs.

2.2. Use Case: Coherent Multi-level Caches

Coherent multi-level caches are functionally simple, although implementation-wise they can be very complex. The bugs usually appear in the coherence protocol, which causes an incorrect value to appear in the system, and eventually be written back to the simulator main memory.

In dynamic testing methodology, we test the following invariants: (1) every read from a location needs to return the last value written (non-speculatively) by any core to the same location, and (2) every write-back from the caches to the simulator memory needs to have the last written value. The given invariants are satisfied at all times, by all types of coherent caches: bus-based, directory-based, broadcast-based or other, with any cache-interconnection topology and interconnection type.

Figure 4 illustrates the application of dynamic testing to coherent multi-level caches. The simple cache emulator consists of a single data structure, an STL map, accessed by all processor cores. The cycle-accurate simulator for the coherent multi-level caches is a collection of C++ objects (one object per cache structure) that: (1) track the access mode and the ownership of the cache lines, (2) track the values (data) of the cache lines, and (3) calculate the access latency of each access. The objects of the cycle-accurate cache simulator communicate by exchanging messages, and each communication increments the total latency of the cache access. For testing the cache access latency, we used an extensive set of unit tests, since dynamic testing tests only the functional correctness.

The memory accesses are sent from the processor to the cache simulator and the emulator. If the cache simulator or the emulator do not have the requested line, they fetch it from the main memory of the simulator, which always has

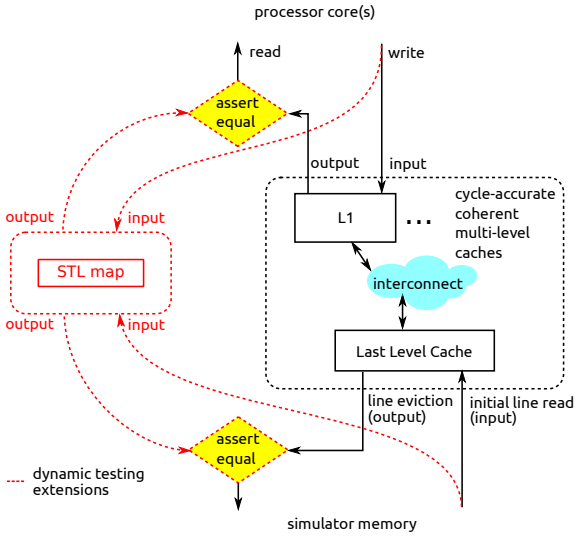

Figure 4. Dynamic runtime testing applied to coherent multi-level caches. The cache lines fetched and evicted by the (1) the cache emulator (STL map) and (2) the cycle-accurate coherent caches, must have the same value.

all cache lines.

In Figure 5, we show the pseudo-code of our implementation of dynamic testing for the cache simulator. A read returns the requested value and checks that the value is the same in both the simulator and the emulator. A write updates the values in two caches without doing any checks. If the cache simulator needs to evict a line, the same location is also removed from the cache emulator, and the data in the two cache lines are checked to be identical. If the data is identical, it is stored in the simulated main memory. Otherwise, the difference is reported to the developer since it indicates a probable bug in the implementation of the coherent multi-level caches. Having the exact point of the execution where the difference appeared, the debugging of the cache-coherence protocol is much simpler.

2.3. Use Case: Out-of-Order Simulator

Dynamic testing can also be applied to the entire cycle-accurate Out-Of-Order (OOO) processor simulator. The biggest problem with the OOO processor simulators are their hard-to-find bugs which occasionally appear, sometimes in particularly long simulations. Worse than that is that these bugs may affect the execution millions of instructions after they happen, thus making debugging almost impossible. Dynamic testing could detect these bugs instantly, as they happen. In Figure 6, we present a schematic

READ

```
// cycle-accurate cache simulator. Multi-level
data = processor[cpuid]->L1->read_data(address, size, &latency);

// cache emulator. Single-level
data_functional = functional_cache.read_data(address, size);
if (data_functional != data) {
   FATAL_ERROR("incorrect cache value: %x instead of %x", data, data_functional);
   // also print the simulator cycle, state, and the accessed address, and then exit
}
```

WRITE

```
// cycle-accurate cache simulator. Multi-level
processor[cpuid]->L1->write_data(address, size, data, &latency);

// cache emulator. Single-level
functional_cache.write_data(address, size, data);
```

LINE EVICT (from the last-level-cache, LLC)

```
// LLC evicts the line with address "address" and data "data"
data_functional = functional_cache.read_data(address, size);
if (data_functional != data) {
   FATAL_ERROR("incorrect data in evicted line address: %x", address);
   // also print the simulator cycle, state, and the accessed address, and then exit
}
```

Figure 5. Pseudocode of the dynamic testing for the coherent multi-level caches

overview of a proposed dynamic testing configuration for the OOO simulators.

This use case is slightly different from previous examples of dynamic testing, since here we have only one input and one output. Still, the dynamic testing can be applied the same way as in the previous examples. To dynamically test the OOO simulator, we can compare its memory writes with the memory writes of a processor emulator. Having identical memory writes during the *entire* simulation provides a strong confidence in the correctness of the cycle-accurate OOO processor simulator.

Since processor emulators are much faster than the cycle-accurate OOO simulators (two or more orders of magnitude), the dynamic testing should not significantly affect the speed of the simulator. Cycle-accurate OOO simulators are inherently slower from the emulators since they simulate the functionality and the interaction of all hardware elements physically present in OOO processors, while processor emulators typically only decode instructions, execute them, and then update the simulated memory.

2.4. Other Use Cases

Similarly to the given examples of dynamic testing, the same principle could be used to improve the functional correctness of other cycle-accurate hardware simulators, and to simplify their debugging without significantly reducing their performance. In general, a tested hardware simulator should evaluate an extension or a modification representable by a simple, functional emulator. For example, dynamic runtime testing can be used for: single-processor multi-level memory hierarchy, incoherent multi-level mem-

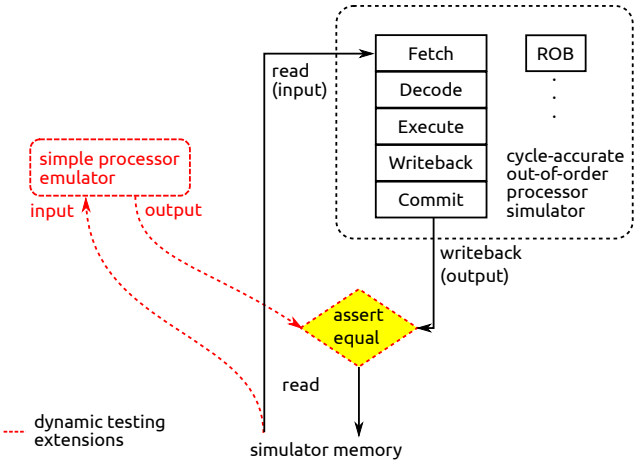

Figure 6. Dynamic runtime testing applied to the entire Out-of-Order simulator.

ory hierarchy, system-on-chip simulators, on-chip routing protocols, or pipelined processors.

3. Finding and Fixing Simulator Bugs

After dynamic testing reports a potential bug in the simulator, a developer needs to find the source of the bug (or debug) the simulator using conventional debugging methods. Common debugging methods are: (1) using a conventional debugging tool, or a debugger, for example *gdb* [12], and (2) execution traces.

Debugger allows a developer to stop the simulator execution at the moment he finds the most appropriate, and then to examine the state of the simulator memory and the architectural registers. This allows the developer to explore the simulator state in details, and even to test the output of particular simulator functions, or to set some memory values manually. Debuggers in general have a property of advancing "forward in time". Unfortunately, this means that if a developer misses the point of failure, he generally has to stop the simulator execution, restart it, and then wait until the execution comes to the same point.

Although not necessary for debugging, a checkpoint support in gdb [13] helps for debugging long simulator executions. Checkpointing allows a developer to save a snapshot of the simulator state, and to restore it later. Restoring a checkpoint rolls back the simulator execution to a point where the checkpoint was made (excluding the input/output operations). Effectively, restoring a checkpoint allows a developer to go back in time and avoid a complete re-execution of the simulator. However, the developer still needs to make a checkpoint at the right moment of execution. Gdb internally implements checkpoints by "fork"ing the execution, and currently the use of checkpoints is not recommended for debugging multi-threaded programs. Since most architecture simulators (including our simulator) are single-threaded, this is not an issue. However, given the increasing popularity of multi-core platforms, the future versions of gdb will likely improve their support for debugging multi-threaded programs.

Trace-based debugging consists of instrumenting the simulator code, in order to print to a trace file a part of simulator context important to the developer. The developer instruments the code while he is developing the simulator. If a developer during debugging realizes that he needs more information from what he has in the trace, he needs to change the simulator code and re-execute the complete simulation. This makes the trace-based debugging somewhat more rigid from a debugger-based debugging, since a developer cannot stop the execution at some point and analyze the simulator execution more deeply. On a positive side, the trace-based debugging naturally allows a developer to analyze application execution forward and back in time.

Traces may contain any amount of information, although a developer needs to take care and make a careful balance between: (1) the readability of traces, (2) the amount of information in the traces, and (3) the size of the traces. Printed trace lines are usually stored in files, to facilitate the posterior analysis of the execution. Due to the common length of simulations, trace files can easily occupy tens or hundreds of gigabytes. Therefore, the developer needs to have a good knowledge of text processing tools, and having fast development machines.

In our development, we typically combined the two debugging methods, by starting with tracing, and switching to the debugger as needed, for more difficult bugs. After our dynamic testing reports a possible bug, that is, an incorrect value of a variable (say, X), we turn on verbose tracing and re-execute the simulation. To find earlier uses of the location X, we analyze the traces from the given execution point backwards. A bug typically occurs in the last access to X, and less frequently it occurs 2-3 accesses before. A bug often becomes obvious after analyzing the memory accesses and the simulator events, and fixing the bug is usually a simple task. To use a debugger, we can make a debugger checkpoint right before the bug, since we know exactly where the bug appeared in the execution. After finding the bug and fixing it, it is desirable to re-run the entire benchmark suite, since fixing the bug might uncover or create other bugs.

4. Evaluation

In this section, we show the performance impact of dynamic testing on simulator performance (execution time). We have used the M5 full-system simulator [2] as a base architectural simulator, and extended it to implemented

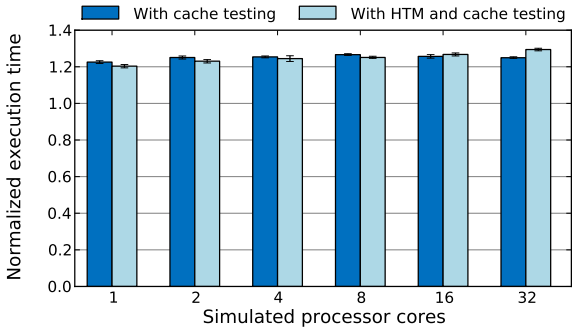

Figure 7. Dynamic testing impact to the simulator speed during Operating System (OS) booting. The average simulator speed is normalized to the one without dynamic testing.

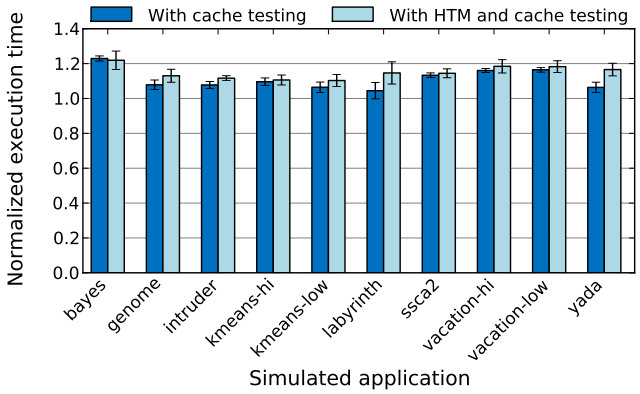

Figure 8. Dynamic testing impact to the simulator speed during application execution. The average simulator speed is normalized to the one without dynamic testing.

MESI-directory coherent multi-level caches, and several HTM proposals.

We carried out all simulations on modern Intel Xeon X86_64 processors, taking care of minimizing the I/O and other system calls, which may non-deterministically affect the simulator performance. As a result, all simulator executions have more than 98% CPU utilization on average. We have measured the execution time of the simulator for all applications from the STAMP transactional benchmark suite [3], and for 1, 2, 4, 8, 16, and 32 simulated processor cores. The simulator is single-threaded, and to simulate multi-core processors, the simulator sequentially processes events of each simulated processor core or device. We have repeated each execution three times to reduce the effect of wrong measurements in single executions caused by random, uncontrollable events, and then calculated an average execution time.

Figure 7 shows the impact to the time needed to simulate the booting of the Operating System. We have grouped the simulator executions by the simulated number of processor cores, normalized the execution time to the simulator without dynamic testing, and then calculated the geometric mean. The results indicate that dynamic testing reduces the simulator speed by 20% on average, with a very small standard deviation. Since there are no transactions during the booting of the OS, there is almost no penalty for doing the empty calls to the HTM testing code.

Figure 8 shows the performance impact of dynamic testing during application execution. We have grouped the simulator executions by the simulated application, normalized the execution time to the simulator without dynamic testing, and then calculated the geometric mean. According to the evaluation, dynamic testing reduces the execution time between 10% and 20%, which is relatively less than during the OS booting. The reason is that the basic simulator is now more complex and simulates an HTM protocol. We can see that, while dynamic HTM testing does introduce some overhead, the total increase in the simulator execution time is generally below 20%.

In both testing examples, dynamic runtime testing would extend a 10 hour simulation to less than 12 hours on average. Taking into account that writing the simulator and the simulator test suite may take many person-months, we consider the performance impact of dynamic testing to be more than acceptable.

5. Our Experience With Dynamic Runtime Testing

It is commonly believed that the earlier a defect is found, the cheaper it is to fix it [9]. Our experience is certainly in accordance with this popular belief. We have developed the dynamic testing methodology out of necessity. Making a cycle-accurate architectural simulator is certainly not easy and, as any other software development, it is very prone to errors.

The original cache coherence protocol in M5 simulator is bus-based, which does not scale well beyond 8 cores (or 16 cores as a maximum). We have replaced the base M5 cache coherence with MESI directory-coherence protocol, known to scale well even with more than 64 processor cores. Our directory-based coherent caches hold both line addresses and data, which means that a bug in the cache-coherence protocol would cause wrong data to be provided by caches. Thanks to using dynamic testing, we were able to complete the implementation of caches in under 3 months, and to have much stronger confidence in the correctness of our im-

plementation.

For our first two HTM simulators, we implemented did not rely on the dynamic testing methodology. The target of the two simulators was validating the results presented by LogTM [10] and TCC [4]. After more than 12 man-months spent on simulator development we had to cancel the development, since some simulations were still not terminating correctly, or were giving wrong results. The execution traces had hundreds of gigabytes, and finding errors in them was nearly impossible.

Dynamic testing methodology in our following simulators allowed us to significantly reduce the time needed to transition from an idea to getting the evaluation results. The benefits from dynamic testing are two-fold. First, since we knew *exactly* where a bug appeared in the simulator execution, we could quickly detect and eliminate all obvious simulator bugs. This reduced the simulator development time from 12-18 man-months to 3-4 man-months. Second, dynamic testing methodology improved our confidence in the results of our evaluations, since we had a proof that our HTM simulators were functionally correct.

Three of our HTM simulators have lazy version management and one has eager version management. Although the functionality of these HTMs is different, they all have similar functional-HTM equivalents. A fundamental difference between the eager and the lazy HTM is the decision on when to abort a conflicting transaction. In both implementations, a transaction can keep its speculatively modified values private, in a per-transaction buffer, and these speculative values can become public when the transaction commits.

6. Conclusions

To increase the stability of cycle-accurate architectural simulators, developers often put at least as much effort in testing and debugging, as in writing the code. Still, errors may occur in the simulators even with the most rigorous testing. Tests rarely cover whole 100% of the source code, and even more rarely 100% of all possible execution paths. The number of the possible combinations of execution paths grows nearly exponentially with the size of the source code (assuming that a number of conditional branches is constant over the source code).

Academic development of architectural simulators is in even more difficult situation than the industrial. In academia, the development teams working on simulators are much smaller than in the industry, and the simulator changes and evaluations are typically done quickly and with short deadlines. This discourages these development teams from writing extensive test suites for the simulators. As a consequence, if the tests exist, they are typically sparse and unsystematic.

Dynamic runtime testing is an alternative approach, where the functional correctness of the simulator is verified automatically with every simulator execution. This allows developers to change the simulator rapidly, and still be able to find bugs quickly and be confident that the simulator executes correctly. The simulator reports any potential bugs, together with the exact time and the circumstances that lead to the bug. The method imposes a minor reduction in simulator performance and, in our case, we have managed to reduce the total time for simulator development and evaluation from 12-18 person-months to 3-4 person-months.

References

[1] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[2] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[3] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multiprocessing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[4] L. Hammond, B. D. Carlstrom, V. Wong, M. Chen, C. Kozyrakis, and K. Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro*, 24(6), Nov-Dec 2004.

[5] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010.

[6] M. J. Harrold. Testing: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 61–72, New York, NY, USA, 2000. ACM.

[7] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[8] R. Jindal and K. Jain. Verification of transaction-level systemc models using rtl testbenches. In *Formal Methods and Models for Co-Design, 2003. MEMOCODE'03. Proceedings. First ACM and IEEE International Conference on*, pages 199–203. IEEE, 2003.

[9] C. Kaner, J. Bach, and B. Pettichord. *Lessons learned in software testing: a context-driven approach*. Wiley, 2002.

[10] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *In proceedings of the HPCA-12*, pages 254–265, Feb. 2006.

[11] P. Runeson. A survey of unit testing practices. *IEEE Softw.*, 23:22–29, July 2006.

[12] R. Stallman and R. Pesch. The gnu source-level debugger. *User Manual, Edition 4.12, for GDB version*, 4.

[13] R. Stallman, R. Pesch, S. Shebs, et al. *Debugging with GDB*. Free Software Foundation, 1993.

[14] M. Vouk. Back-to-back testing. *Information and software technology*, 32(1):34–45, 1990.