Open access • Proceedings Article • DOI:10.1109/ISPASS.2009.4919656

QUICK: A flexible full-system functional model — Source link

Dam Sunwoo, Joonsoo Kim, Derek Chiou

Institutions: University of Texas at Austin

Published on: 26 Apr 2009 - International Symposium on Performance Analysis of Systems and Software

Topics: PowerPC and Out-of-order execution

QUICK: A Flexible Full-System Functional Model

Dam Sunwoo, Joonsoo Kim and Derek Chiou
The University of Texas at Austin
{sunwoo,turo,derek}@ece.utexas.edu

Abstract

In this paper, we introduce the concept of full-system Complete-and-Rollback functional simulators that make efficient functional models in functional/timing partitioned simulators. Complete-and-Rollback functional simulators can efficiently drive simulators of resolutions ranging from functional-only to cycle-accurate for a wide range of simulated machines. Complete-and-Rollback functional models achieve their capabilities by executing instructions to completion, enabling their execution to be highly optimized, but providing rollback capabilities to enable on-the-fly modifications to the functional execution.

We also introduce QUICK, an implementation of a full-system Complete-and-Rollback functional model that supports the x86 and PowerPC ISAs, boots unmodified Windows XP and Linux, and runs unmodified applications such as YouTube on Internet Explorer while fully supporting rollbacks, including across I/O operations. We present various case studies using QUICK and conduct performance analyses to demonstrate its simulation performance.

1. Introduction

Due to the high cost and time consuming nature of designing, testing, and manufacturing computer systems, simulation is extensively used to model and predict various attributes, such as performance or power, throughout the design cycles. Developing such simulators often requires a great amount of effort to achieve high accuracy and efficiency. The already high cost of development kept many simulators from supporting full-system and complex ISAs such as x86 at the cycle-accurate level, even when such features will allow the simulators to run a wider variety of benchmarks.

Simulators are often partitioned into functional and timing models[6, 12, 11, 15, 2, 3]. The functional model(FM) simulates the ISA and the functionality of peripherals, whereas the timing model(TM) models the detailed timing of the microarchitectural components. The two partitions cooperate to implement the entire simulator functionality. By partitioning what tends to change very little, the target functionality, from what tends to change a lot, the microarchitecture, implementation effort can be dramatically reduced compared to a simulator that does not partition on this boundary. Once the FM is developed, it can be reused virtually indefinitely, making it worthwhile to spend a significant effort on its development.

There are several variants of functional/timing partitioned simulators. One important difference is how the two partitions collaborate to implement the full simulator. At one end of the spectrum is an execution-generated trace-driven simulator, where the FM executes instructions and generates a trace that is fed to the TM. In this case, the FM does not change its execution in response to the TM. A variety of high-performance techniques can be applied to generate such a trace, for example, using dynamic binary instrumentation or modifying just-in-time compiled functional simulators, such as Simics[7] or QEMU[1]. Such a simulator is simple, but cannot always be accurate since the TM may not get all the information it needs from the FM. For example, if the target[1] machine mispredicts a branch, the TM will not have the *correct* wrong-path instructions.

On the other end of the spectrum, the FM can be seen as a set of subroutines that the TM calls to perform various functional tasks at precisely the correct simulated time. For example, the Asim[6] FM is split into Fetch, Decode, Execute, Memory, Commit, and WriteMemory stages, each of which is called by the timing model to perform its specific task on a specific set of arguments at the specific simulated time it would have been processed. Each FM subroutine executes independently and thus must save sufficient state when it finishes so that the next logical subroutine may be executed at any time in the future. Also, the FM must logically support the same sort of structures that the target microarchitecture supports. For example, if the target has a reorder buffer (ROB) that permits it to cancel the execution of certain instructions, such a structure must also be available in the FM. We call such an FM a *timing-driven* FM.

[1] We use the term *target* to mean the machine that is being simulated and the term *host* to mean the machine that runs the simulator.

A middle ground strategy is an FM that fully executes instructions and generates a trace of those instructions, but also provides the ability to roll back to a past instruction and restart execution with potentially different state, such as a different branch outcome. We call such functional models *Complete-and-Rollback*. Such an FM provides a trace of the instructions it executes in the order it chooses to the TM that determines whether that stream of instructions is the stream of instructions it would have fetched and executed in that order. The TM can instruct the FM to change its instruction path using the rollback/change state ability. Complete-and-Rollback FMs enable *simulator-level* speculation, where the FM can run ahead of the TM and be steered by the TM only when it has gone off-path. Therefore, the TM only needs to give feedback to the FM when necessary instead of directing it at every cycle or every instruction, thus dramatically improving performance and lowering the design complexity of the overall simulator.

Rolling back and correcting is a general way to model an arbitrary target with a generic FM, but does incur the overhead of enabling and performing the rollbacks. However, since target microarchitectures are generally designed to provide the illusion that they execute instructions in the same way as an FM of that target, rollbacks should be infrequent. In fact, the closer the target is to the functionality it is intended to support, the fewer the rollbacks.

In this paper, we introduce QUICK(QemU with Instrumentation and ChecKpointing), a Complete-and-Rollback full-system simulator that supports both the x86 and PowerPC ISAs, a full set of peripherals, and boots and runs unmodified operating systems such as Windows XP and Linux. Because QUICK is derived from an existing open-source full-system functional simulator, QEMU[1], it inherits QEMU's extensive current and future ISA/peripheral support. QUICK produces a trace suitable for on-line export to a TM and supports rollback and re-execution, even across I/O operations.

Our contributions in this paper are as follows:

- We introduce the notion of a Complete-and-Rollback functional model that is useful for both (i) modeling a wide range of target architectures at various resolutions and (ii) to implement simulator speculation.

- We discuss and analyze mechanisms that enable the FM to arbitrarily roll back and steer the instruction stream, even across I/O operations.

- We present QUICK, a real implementation of such an FM.

- We demonstrate the usefulness and performance of such an FM with several case studies and performance analyses.

The rest of the paper is structured as follows. The next section describes how an FM that supports speculation and correction can simulate a wide range of target systems. Section 3 describes the implementation of QUICK. Section 4 presents some experiments and performance measurements. Section 5 lists related work. Section 6 discusses future directions and concludes.

2. Complete-and-Rollback Functional Models

It is desirable for a single FM to be able to model as many different targets as possible. A Complete-and-Rollback FM can be used to model a wide range of target machines, even though those machines may have very different characteristics than the FM itself.

A Complete-and-Rollback FM executes each instruction to completion but provides the ability to roll back to any instruction within a rollback window, change state, and re-execute. Supporting target speculation is straightforward with a Complete-and-Rollback FM. Figure 1 shows how branch misprediction is handled. In Figure 1(a), the FM executes down the right-path, sending traces to the TM. The TM discovers that instruction `B`, that happened to be a branch instruction, was mispredicted and should have branched to `X`. The TM discards the trace entries of instructions `C` and `D` since they are not expected by the TM. In Figure 1(b), the FM rolls back to instruction `B`, and forces a branch to `X`. The TM now starts accepting traces for the wrong-path. The mispredicted branch, `B`, is resolved in the TM at some point and the FM rolls back, leading to Figure 1(c). The FM re-executes the right-path again and the traces are accepted by the TM. The final instruction sequence seen by the TM (A→B→X→Y→C→D) represents the instructions that have entered the pipeline of the target processor.

Data speculation can be handled in a similar fashion. The FM executes as it would normally. When the TM detects that the FM executed an instruction with *incorrect* data, it requests the FM roll back to that instruction and re-execute it with the *correct*[2] misspeculated data. Likewise, when the TM detects that the data was misspeculated, it requests the FM roll back and re-execute with the right data.

Although an ideal FM requires the capability to arbitrarily roll back, implementing such capability is a non-trivial task as the FM may need to roll back across host I/O operations (if the simulator I/O is tied to host I/O devices as is the case with many current full-system simulators.) When the FM encounters a target I/O operation, it cannot wait until the TM determines the fate of the operation as the TM will

[2] We use the term *correct* to indicate all of the instructions that the target would fetch and execute, including *right* and *wrong* path instructions. *Incorrect* instructions are those executed by the FM that the target would not have fetched/executed (at least not in the incorrect order.)

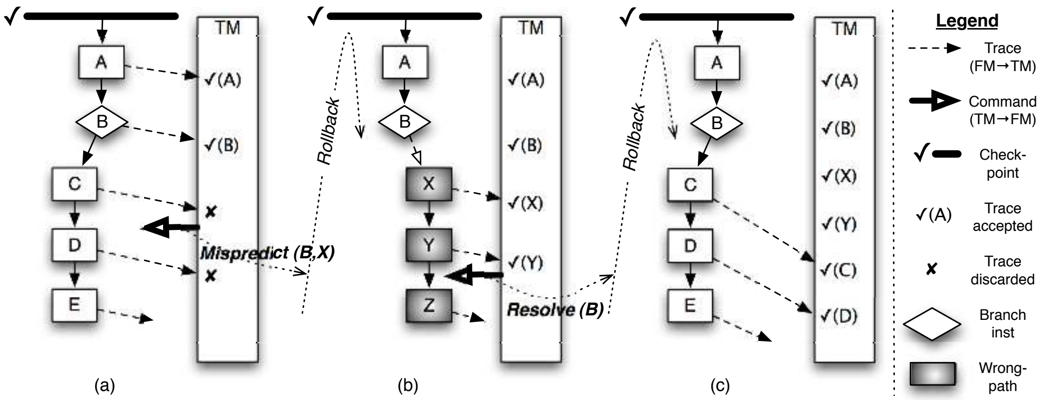


Figure 1. Example of handling branch mispredictions

require subsequent instructions to make forward progress. This issue will be discussed in detail in Section 3.1.2.

In a split functional/timing simulator, the FM both executes the workload and provides functional information to the TM. For example, the TM requires instruction information to accurately model Decode functionality and the opcode type to accurately determine which functional unit. However, many targets perform different components of the functionality of instructions in different orders, depending on the dynamic conditions of the processor. For example, two instructions may be executed in a different order than they are fetched. Even a simple in-order pipeline often has a non-blocking cache that effectively executes memory operations out-of-order. Thus, a general FM must be able to support arbitrary orderings of functional components across instructions.

Almost all reorderings of functional components, however, do not affect functionality. For example, a standard out-of-order target can execute instructions in any order, as long as their data dependencies are obeyed. However, any instruction order that maintains data dependencies will have the same functional result as any other. Consequently, the instruction order, that most FMs will follow, will produce the correct result and will, given appropriate rollbacks to model misspeculation, provide the correct instructions so that the TM can accurately predict performance.

To summarize, a Complete-and-Rollback FM needs the following capabilities: functional simulation, trace generation and the capability to roll back and steer instruction stream arbitrarily, even across host I/O events. Such an FM can support both trace-driven and execute-driven simulation ranging from a functional-only simulation to a cycle-accurate simulation of a complex target system.

3. QUICK

QUICK is a full-system Complete-and-Rollback FM that uses aggressive just-in-time compilation to generate native code to very quickly simulate target instructions. To creak QUICK, we extensively modified QEMU to incorporate trace generation and rollback capability, features not normally desired in a high speed functional simulator. In this section we describe those modifications to QEMU while maintaining full-system support and reasonable performance.

3.1. Rollback Support

QUICK has the capability to roll back to an instruction that was already executed, removing all effects of the instructions as it rolls back. In this section, we describe how rollback is implemented (i) when running normally, (ii) when accessing I/O and (iii) in the presence of interrupts and exceptions.

3.1.1 Basic Implementation

Since QUICK is intended to support cycle-accurate simulation, it must be able to roll back very quickly within a rollback window of up to a few thousand instructions. Standard checkpointing, that saves the entire state of the simulator to disk, is far too slow. Saving a full checkpoint to memory would both require too much memory and would also take too long since the target physical memory could be many gigabytes.

Rather than full checkpoints, rollback can be supported by logging old values from each state update in a journal. On rollback, the old values are restored in reverse order. Journaling works well for the memory where the fraction of locations that changes within a rollback window is very small compared to the total memory size. Since registers tend to change much more rapidly and the fraction of the registers that change within a rollback window is very high, checkpoints work better for the CPU and register state. For this reason, QUICK employs a hybrid scheme of journaling and full-state checkpoints. Only the CPU architectural state (general and special purpose registers) are included in the checkpoint, making its size only a few kilobytes for the x86 ISA.

It is clear how to rollback to a specific instruction given a journal back to that instruction. With a checkpoint, one can roll back to the instruction immediately before the checkpoint was taken by restoring the checkpoint. In order to get to an arbitrary instruction using checkpoints, one has to roll back to the checkpoint before the desired instruction and then re-execute instructions up to the desired instruction. QUICK supports rollback to any instruction within the instruction window transparently.

As instructions fall out of the rollback window, such as when instructions are retired by the TM, journal/checkpoint state can be deallocated. It is clear when journaled data can be deallocated. Checkpoints can only be deallocated when there is at least one checkpoint that occurred before an instruction that is either beyond the rollback window or the oldest instruction in the rollback window. Therefore, at least two checkpoints are necessary, one to enable rollback and the other to be deallocated. QUICK implements at least two checkpoints that are taken in a leapfrog fashion, one after the other.

3.1.2 I/O Rollbacks

Full-system functional simulators include functional I/O device models, implemented in software, that provide precisely the same interfaces that the real devices would. The target device drivers interface with those models as they would with real hardware devices. The device models call the real devices, through the host operating system, to implement their functionality. Since a functional-only simulator does not roll back, the I/O device models immediately perform I/O operations to the host I/O in response to commands it receives.

A Complete-and-Rollback simulator, however, cannot eagerly perform host I/O operation since one cannot normally roll back host I/O operations. One possible way to handle such a case is to defer a host I/O operation when it is first encountered, execute the subsequent operations, and roll back and execute the I/O operation when it commits. This solution, however, not only requires a rollback for every single I/O operation, even on the correct path, but may also require the FM to patch traces already sent to the TM as the traces may have contained information dependent on the I/O operation.

A more efficient solution is to let the target I/O operation execute and roll back when necessary. If the I/O operation was on the correct path, which is true most of the time given accurate branch predictors, no further action is required. However, if the target I/O operation interacts with a host I/O operation, it may be impossible to roll back the effects. For example, there is no way to retrieve an Ethernet packet once it is actually sent from the host Ethernet device. When a target event triggered by a host-level keystroke is rolled back due to a branch misprediction, the same keystroke will not occur on the re-execution and the input would be lost if not handled properly.

To address this issue, QUICK executes that operation against the host when the target requests an input I/O operation for the first time. Thus, in the case of keyboard input, the next keystroke will be dequeued and provided to the target. However, to support arbitrary rollback, the input operations need to be logged so that they can be *replayed* exactly at the same target time to provide deterministic behavior in the case of rollback. To ensure determinism in the case of an I/O input operation that occurs while re-executing or on the wrong-path, the input operation is deferred until QUICK reaches an undiscovered path that was never re-executed.

QUICK handles host output I/O operations by buffering outputs to host devices, such as Ethernet outputs or serial console outputs, since host devices cannot be rolled back. The buffered entry is released when the operation is committed by the TM and discarded on rollbacks. Buffering the host I/O operations can affect the short-term state of the target. For example, an input operation may immediately follow an output operation the input may depend on in a hypothetical infinitely-fast device. In such a case, one could employ special simulator structures similar to a Load-Store-Queue to forward the uncommitted state changes. Realistically, however, the latency of I/O operations is usually longer than the time instructions take to commit. Subsequent I/O operations would happen after the previous operation has completed in most cases.

QUICK provides these input replay/output buffering capabilities for all major I/O devices, including keyboard, mouse, serial ports, video, disk and network. Providing such capabilities enables a much wider range of interesting applications to be run.

The host-level input operations from devices can be saved along with the target timestamp. This allows the user to *record* the I/O events of one simulation session and *play them back* during another for determinism across simulation runs. This mechanism is especially useful if the user records an interactive session in a fast-forward mode without timing simulation, and then uses the recorded session in a detailed cycle-accurate timing simulation.

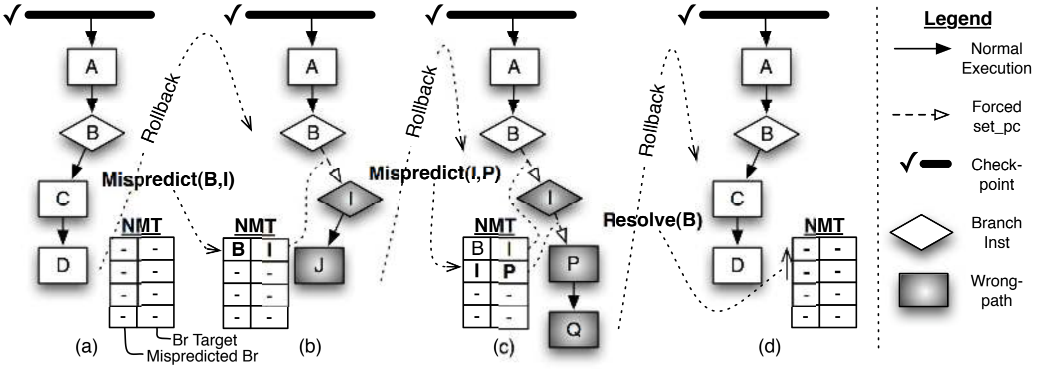


Figure 2. Example of handling nested branch mispredictions

3.1.3 Nested Rollbacks

As the depth of the pipeline becomes deeper in modern processors, multiple branch mispredictions can be outstanding before any of them gets resolved. Branches can also be resolved out-of-order. When simulating such cases, QUICK needs to be rolled back several times and is expected to steer itself down the target-selected wrong-path.

Figure 2 shows an example of nested branch mispredictions. QUICK has already executed to instruction `D`. After the TM notifies QUICK that the target mispredicted the branch at instruction `B`, QUICK rolls back to the closest checkpoint (right before `A`) and provides the correct wrong-path instruction, a branch at `I`. Then, the TM mispredicts `I` as well. In response, QUICK rolls back once more to the same point, but must remember to go down the same wrong-path when re-executing `B` again.

A simple solution to ensure that `B` is misspeculated on the re-execution to get back to instruction `I` is to do an associative search across for all currently misspeculated branch instructions to decide which path to take. This is very inefficient, especially in a software functional model. To handle branch misprediction, re-execution, and resolution efficiently, QUICK incorporates a Nested Misprediction Table (NMT) to store the outstanding mispredicted `inst_num`[3] and desired target address pairs.

The NMT has a *write* pointer that points to the next empty entry. On every branch misprediction, the mispredicted `inst_num` and target address pair is written into the entry pointed to by the write pointer. QUICK then enters a *speculative* mode that disables taking checkpoints and also sets the *read* pointer to the first misspeculated branch stored in the NMT. As QUICK re-executes while in speculative mode, it compares every branch `inst_num` with the `inst_num` pointed to by the read pointer. If a match is found, QUICK forces the PC down the corresponding target and advances the NMT read pointer. If the current `inst_num` is greater than the `inst_num` at the read pointer, QUICK knows that a non-branch was misspeculated and recompiles to break the basic block at that `inst_num` to permit that instruction to be correctly re-executed/resolved. When a previously mispredicted branch is resolved, the entries in the NMT for the corresponding branch and onward are invalidated by setting the write pointer appropriately. QUICK does not exit speculative mode until the first branch in the NMT is resolved.

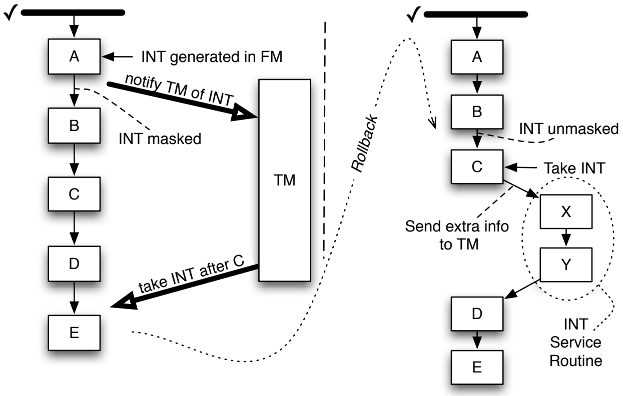


Figure 3. Example of handling interrupts

In Figure 2, when branch instructions `B` and `I` are mispredicted one after the other, the NMT is updated accordingly. When instruction `B` is resolved, the NMT is cleared. If instruction `I` is resolved sooner than `B`, causing an out-of-order resolution, only the second entry in the NMT is invalidated. The NMT reduces an associative search to correctly perform re-execution and out-of-order branch resolution down to a single comparison.

[3] QUICK dynamically numbers every executed instruction with an `inst_num` that is rolled back during a rollback.

3.1.4 Handling Interrupts and Exceptions

Interrupts and exceptions[14] need special attention in QUICK for the following reasons:

1. Interrupts are, by definition, external events, whose occurrence is *timing dependent*. The TM is responsible for calculating the exact timing of the interrupts. Thus, for TM-initiated interrupts, like the timer interrupt, the TM simply requests the FM to roll back to the exact instruction where the interrupt occurred and handle the interrupt there. However, since the FM models some of the interrupts, mostly peripherals, there needs to be a mechanism for the FM to notify the TM of a pending interrupt and the TM to indicate to the FM exactly when that interrupt occurs.

2. Many targets detect exceptions when the excepting instruction retires. The TM generally needs additional target path instructions to fill up the front of the pipeline. A standard functional simulator, however, immediately detects the exception and jumps to the exception handler routine. The FM needs to be able to provide the target path instructions to the TM, effectively ignoring the exception until the TM indicates it should be noticed.

Figure 3 shows the proposed mechanism to handle interrupts correctly. When QUICK detects a pending interrupt in one of its peripheral device models, it masks the interrupt and notifies the TM. QUICK proceeds on its original path until the TM asserts the interrupt back to QUICK. The TM provides the `inst_num` at which the interrupt should be serviced. On receiving such a command, QUICK rolls back to the closest checkpoint and re-executes. In this example, the TM requests QUICK to service the interrupt after instruction `C`. After rolling back, QUICK re-executes up to instruction `B` and unmasks the interrupt. Before jumping to the interrupt service routine, QUICK may need to send the TM extra information regarding any memory accesses or processor updates that have occurred while processing the interrupt prior to the service routine. If the interrupt service point (from the TM) is in the middle of a basic block, QUICK splits the block in two by re-translating that block within QEMU, since interrupts are currently only triggered on basic-block boundaries. QUICK tracks the instruction counts in each block during the first pass to exactly identify which block to re-translate.

Other branch mispredictions may occur after jumping to the interrupt handler but before committing the current checkpoint. After rolling back on such mispredictions, QUICK needs to be able to *replay* the interrupts independently as the TM will not issue the interrupt command again. By splitting basic blocks to signal the interrupts at the appropriate place, QUICK deterministically replays the interrupts. In most microarchitectures, interrupts are not handled on a wrong-path since interrupts are handled after draining the pipeline and waiting until all previous instructions are committed or flushed.

To generate target timer interrupts, QUICK features a virtual timer instead of the original QEMU real-time timer to ensure determinism. The virtual timer is based on instruction count. As long as the timing of external events is reproduced identically, perfect determinism can be achieved. Ideally, timer interrupts should be initiated by the TM as the TM is the master of time. The user can choose to put a timer model in the TM for perfect cycle-accuracy or use QUICK's virtual timer for a deterministic approximation.

Exceptions are handled very similar to interrupts. Exceptions are also masked and deferred until the TM discovers them at the right moment, after additional instructions on the original path are sent to the TM. Not all exceptions require these additional instructions to be sent. Exceptions such as instruction page faults or invalid opcode exceptions make it impossible to fetch subsequent instructions and, hence, no additional instructions are needed from the FM. On subsequent rollbacks due to mispredictions, the exception does not need to be replayed (unlike interrupts) since it will be detected naturally in QUICK.

3.2. Trace Support

The QUICK trace includes opcode, source and destination register usage, instruction and data addresses (both virtual and physical), instruction length (for variable-length x86 instructions) and branch information. An x86 instruction can be expressed in eight 32-bit words including all information mentioned above. Since QUICK has already decoded the instruction, it sends a flattened 10-bit opcode to ease the decode complexity in the TM. To support tracing, new tracing micro-ops were added to QEMU and are introduced during QEMU translation.

Instruction trace entries contain static trace elements and dynamic traces elements. Static traces entries do not change with each dynamic instance of an instruction, including opcode and register information. Dynamic trace entries may change every time an instruction is executed, such as data addresses and branch information (taken or not-taken). To minimize computational overhead, the static trace is computed at translate time and stored away as constant values in the translated code. Only the dynamic trace is generated during execute time and appended to the static trace before being sent to the TM. Generated traces can be dumped to a file for offline usage, passed through a TM function call, or written to a circular buffer allocated in a memory region shared to use with a TM in a separate process.

Since the trace takes up most of the communication bandwidth between the FM and the TM, compressing the trace can improve performance and make the system compatible with a wider range of hosts. Data and instruction addresses take up four out of the eight words. We developed a compression technique that relies on the native translation TLB that QEMU implements for its own internal translation. By mirroring the QEMU TLB in the TM and sending updates to QEMU TLB to the TM TLB, the TM can translate virtual addresses to physical addresses, eliminating the need to send the physical address.

We also compressed the instruction addresses. As long as the instruction does not branch, the instruction virtual address can be obtained from the address and size of the previous instruction. The instruction size is already part of the trace to support accurate simulation. Thus, only the starting virtual address of a basic block needs to be sent to the TM. These optimizations reduce the trace size of an instruction to less than three 32-bit words on average.

The trace size can also be further reduced by caching static trace entries in the TM. The next time a block is executed, only the block ID, along with the necessary dynamic information, is sent. It is important to tag the dynamic data. Host TLB updates might occur before any memory access. A data memory access may or may not occur for certain instruction including x86 repeat string operations and predicated instructions, depending on the state of the target. An exception may occur in the middle of a basic block, causing the block to terminate early and jump to the exception handling routine. To handle these situations appropriately when the static trace is cached and not sent again, a short tag with such information precedes data addresses. The trace receiver is responsible for reading the tag first and taking the correct action.

Experiments on the SPEC2000 integer benchmarks indicate that using a static trace cache in the TM has a potential of reducing the total trace size by an additional three times, making the average trace entry less than 32 bits.

4. Case Studies and Simulation Performance

The experiments in this section were conducted on a quad-core Intel Xeon X3230 running at 2.66GHz with 4MB L2 cache per pair of cores and 4GB main memory.

4.1. QUICK+Dinero: Measuring OS Effects on Cache Behaviors

To demonstrate QUICK's flexibility, we attached it to Dinero IV[5], a trace-driven cache simulator. The integration process only took a few hours. To eliminate either a pipe between two processes or a trace file, and thus maximize performance, the two frameworks were integrated into a single process. We used QUICK+Dinero to measure the impact of operating system on cache behaviors of applications. Cache configurations similar to that of an Intel Core 2 Duo described in Table 1 were used. The target processor is configured to have 512MB of main memory. QUICK has the ability to separate operating system code and user code and mark the memory accesses accordingly. The x86 Current Privilege Level (CPL)[9] is used to separate system code from user code. If the CPL is 0 when a certain instruction is executed, we assume the instruction is operating system code. From the generated trace, which contains the full information of an instruction including opcodes and registers, only the relevant information (physical address and access type) is passed to Dinero. Access types include instruction, data read and data write.

L1 I-Cache	32KB, 32B lines, 8-way set associative, LRU
L1 D-Cache	32KB, 32B lines, 8-way set associative, LRU
L2 Cache	Unified 2MB, 64B lines, 16-way set associative, LRU

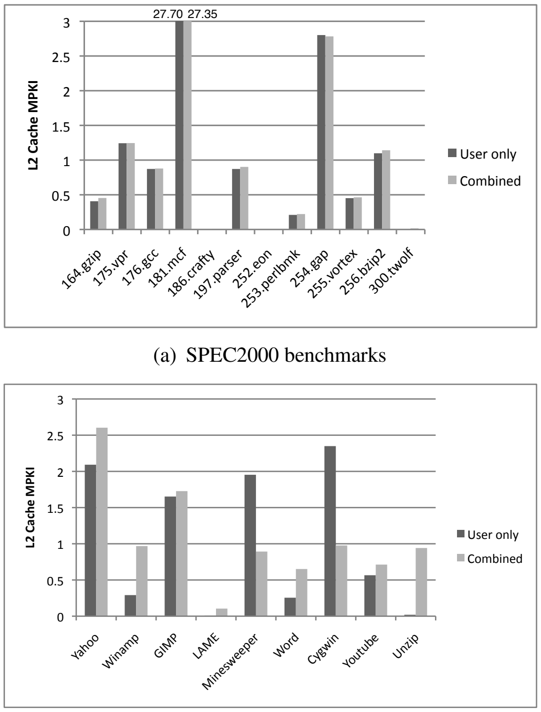
Table 1. Cache configuration used in experiment

Benchmark	Description
Yahoo	Launching Internet Explorer and opening Yahoo.com
Winamp	Installing Winamp 5.54
GIMP	Applying a line nova filter to an JPEG image (1000x667)
LAME	WAV to MP3 encoding of a 3:52 song
Minesweeper	Playing a beginner-level Minesweeper game
Word	Typing in a page-long document in Microsoft Word 2003
Cygwin	Installing Cygwin with default settings
Youtube	Watching a 1-min-long Youtube video on Internet Explorer
Unzip	Extracting a 43MB zip file in Explorer

Table 2. Windows XP benchmarks

In this experiment, two separate instances of the cache hierarchy were instantiated. One hierarchy was fed with both user and OS accesses, whereas the other hierarchy was fed only with user accesses. Among the various cache statistics were gathered, we have chosen to compare the number of cache misses per thousand instructions (MPKI) between the two hierarchies to measure the impact of OS accesses.

Figure 4(a) shows the L2 cache MPKI for the SPEC2000 integer benchmarks running the reference input sets on top of Debian 4.0. The entire set of runs took less than seven days on a single host system without sampling. Both user-only and user/OS access statistics were gathered simultaneously. As is well known, operating system accesses have virtually no impact on the cache when running SPEC benchmarks. We also ran various user applications on Windows XP while simultaneously gathering cache statistics, including watching a YouTube video in Internet Explorer. Table 2 summarizes the Windows user applications that were run for the experiment. L2 cache MPKI numbers are shown in Figure 4(b) for both user and user/OS combined cache hi-



(a) SPEC2000 benchmarks

(b) Windows XP benchmarks

Figure 4. L2 Cache Misses Per Kilo Instructions (MPKI)

erarchies. Unlike SPEC benchmarks, the operating system accesses have a significant effect on L2 cache MPKI across Windows XP benchmarks.

QUICK is also capable of gathering phase behaviors of metrics by periodically polling counters. Figure 5 shows the L2 cache miss rates while booting up Windows XP to the desktop. Miss rates for user and OS accesses are separately shown. Figures 6(a) and 6(b) shows the simulation performance for the QUICK+Dinero simulator running the SPEC2000 and the Windows benchmarks, respectively. Each run is simultaneously simulating two separate cache hierarchies (one with only user accesses and the other with both user and OS accesses.)

4.2. QUICK+BranchPredictor

Although writing a complex and realistic TM for QUICK is beyond the scope of this paper, we present performance results of QUICK with simple TMs that have synthetic target branch predictors to show the overhead of rollbacks. We have run the SPEC2000 benchmarks against TMs with three different branch prediction accuracies: 95%, 97% and 100%. The simulation performance includes the time executing the instructions, generating and sending traces, tak-

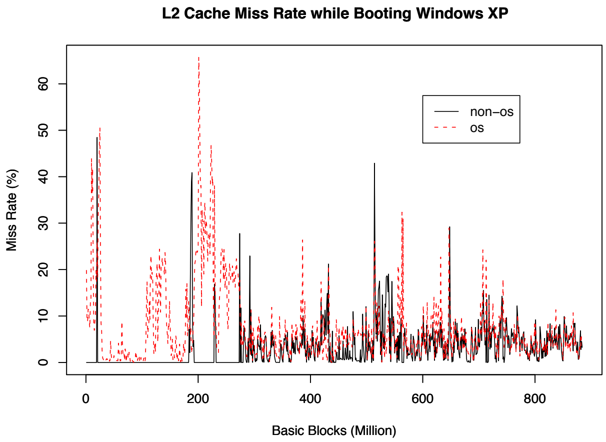
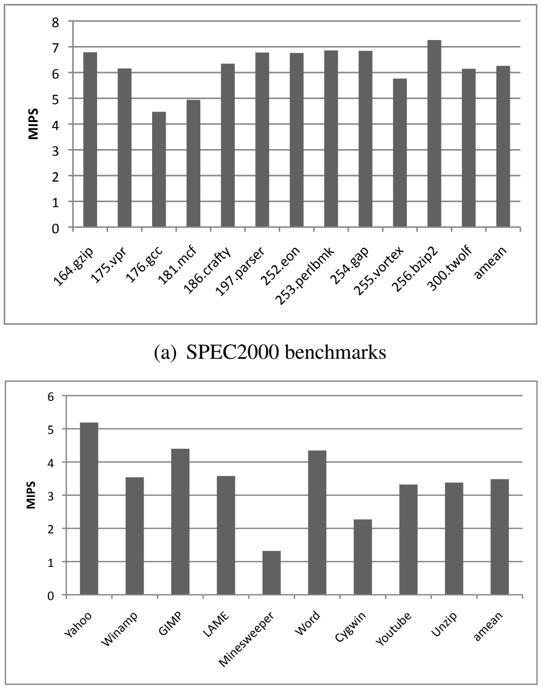


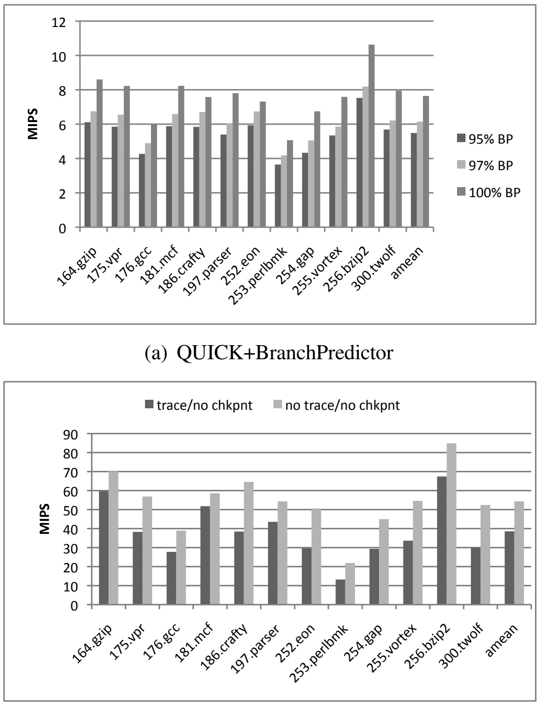
Figure 5. Phase Behavior of L2 Cache Miss Rates during Windows XP Boot



(a) SPEC2000 benchmarks

(b) Windows XP benchmarks

Figure 6. QUICK+Dinero Simulation Speed



(a) QUICK+BranchPredictor

(b) Speed without checkpoints

Figure 7. QUICK Simulation Speeds

ing checkpoints and rolling back to provide wrong-path instructions to the TM. For each mispredicted branch, ten instructions are fetched into the target pipeline before the branch gets resolved.

Figure 7(a) show the simulation performance of QUICK running with the three different branch predictor configurations. The average performances for the 95%, 97% and 100% branch predictors are 5.48 MIPS, 6.15 MIPS and 7.64 MIPS respectively. An analytical model of the QUICK performance could be constructed as,

$$Performance = \frac{1}{T_{inst} + F_{rb} \times \alpha} \quad (1)$$

where T_{inst} is the average execution time for a single target instruction, including the time to generate the trace and take checkpoints, F_{rb} is the frequency of rollbacks, and α is the average overhead of rolling back. From the 100% branch predictor case, $F_{rb} = 0$ and T_{inst} is approximately 131ns. Since a branch misprediction incurs a double rollback, $F_{rb} = 0.06$ for the 97% branch predictor. In this case, α becomes 527ns. If we use these numbers for the 95% case with $F_{rb} = 0.1$, the performance is estimated to be 5.44 MIPS which is very close to the measured performance of 5.48 MIPS. α is only four times larger than T_{inst}. This is due to the fact that, on the re-execution path, checkpoints are not taken and traces are not generated.

If the user decides not to simulate wrong-path instructions, simulation performance can be dramatically increased. The simulation performance without checkpoints averages 38MIPS (Figure 7(b).) The simulation performance without traces and checkpoints averages 54 MIPS, which is slower than unmodified QEMU since some QEMU optimizations, including block chaining and assembly-code MMU, were disabled for easier modifications. These optimizations will be re-enabled in the future.

5. Related Work

SimpleScalar[15] has been widely used in academia since its introduction. The lack of full-system support, however, makes it a less attractive option in modern computer architecture research.

FastSim[12] is a functional/timing partitioned simulator that uses a Complete-and-Rollback FM created from instrumented binary code. FastSim is, however, not full-system since instrumenting operating system code is difficult and thus does not handle the various full-system issues that QUICK handles. In addition, although both QUICK and FastSim employ some sort of checkpoint and rollback scheme, QUICK is different from FastSim because it is able to speculate at the *simulator* level. FastSim calls the target branch predictor function at every branch, which limits the parallelism between the FM and TM[4]. In QUICK, the FM is allowed to freely run ahead of the TM, making it much more efficient to parallelize than in FastSim. The disadvantage is two rollbacks are often required when only one is required in FastSim. QUICK can also model a branch predictor that mispredicts an instruction is a branch.

There is currently a proliferation of fast, full-system functional simulators. Most of them apply dynamic translation techniques that let them run significantly faster than traditional interpreters. Simics[7] and SimNow[13] are not open-source, and thus are difficult to use as FMs that require modifications to handle feedback from TMs. QEMU[1] and Bochs[10], on the other hand, are open-source and lend themselves well to being modified into a Complete-and-Rollback simulator. None of these simulators provide lightweight rollback capabilities in their original forms.

PTLsim[16] is a recent cycle-accurate full-system x86 simulation. PTLsim is highly optimized and runs faster (about 270KIPS) than other simulators at similar detail levels. However, it is not partitioned into functional/timing models, making existing timing models, especially if developed for a different ISA, harder to be integrated and making more abstract studies, such as perfect branch prediction,

[4] Note that, ideally, FastSim should access the branch predictor for every instruction to model a BTB that may incorrectly predict any instruction to be a taken branch.

difficult. Also, maintaining performance after user modifications has been found to be non-trivial.

COTSon[8] employs SimNow as a front-end and PTLsim as a detailed timing model. It uses sampling to determine when to switch back and forth between SimNow and PTLsim. While sampling may improve the simulation performance, it is possible to miss an interesting simulation point if it is not detected as a *phase change*. Also COTSon shares the same downsides with PTLsim of having a functionality-integrated timing model.

Asim[6] and HAsim[4] have the functional/timing split, but require the FM to support the same structures that the target microarchitecture supports. TFsim[11] also has the functional/timing split but the TM still *executes* operations functionally and compares the results with the FM.

FAST simulators[3] recently employed a Complete-and-Rollback partitioning and achieved high performance simulation by placing the TM in an FPGA. In this paper, we generalized this approach to make it compatible with a wider range of host platforms and timing models.

6. Future Directions and Conclusions

QUICK can be extended to simulate multi-processor targets from functional only to full cycle-accurate. Support for detecting and correcting out-of-order memory operations is currently being added. In addition, QUICK is being parallelized so it could run on multiple host cores, providing even more performance when modeling multiple target cores.

To conclude, QUICK, the first generic version of a Complete-and-Rollback FM we are aware of, is presented. It is capable of very different predictive capabilities, ranging from functional-only all the way to complex target cycle-accurate. Simulation performance is excellent at any given predictive level. It achieves its performance by leveraging the power of the optimized just-in-time compilation capabilities of QEMU and efficiently adding trace and rollback capabilities. Such mechanisms greatly simplify the interface between functional and timing models. By enabling simulator-level speculation, they will potentially improve the overall simulation performance when run on parallel hosts.

7. Acknowledgments

We acknowledge Fabrice Bellard and the other developers of QEMU for their very capable open source full-system functional simulator. We thank Nikhil Patil, Hari Angepat, Eric Johnson and Chang Joo Lee for valuable comments and discussions. We also thank the anonymous reviewers for their feedback. This paper is based upon work supported by the National Science Foundation under Grant No. 0615352, a Department of Energy Early Career Principal Investigator Award, IBM Faculty Awards, Intel and Freescale.

References

[1] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX 2005 Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[2] 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.

[3] 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.

[4] N. Dave, M. Pellauer, Arvind, and J. Emer. Implementing a Functional/Timing Partitioned Microprocessor Simulator with an FPGA. In *Proceedings of the Workshop on Architecture Research using FPGA Platforms, held at HPCA-12*, Feb. 2006.

[5] J. Edler and M. D. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator. *web site: http://www.cs.wisc.edu/ markhill/DineroIV*.

[6] J. Emer, P. Ahuja, E. Borch, A. Klauser, C. K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *Computer*, 35(2):68–76, 2002.

[7] P. S. M. et al. Simics: A Full System Simulation Platform. *Computer*, pages 50–58, Feb. 2002.

[8] A. Falcon, P. Faraboschi, and D. Ortega. Combining Simulation and Virtualization through Dynamic Sampling. In *Proceedings of ISPASS*, Apr. 2007.

[9] Intel, www.intel.com/design/processor/manuals/253668.pdf. *Intel 64 and IA-32 Architecture Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*, 2007.

[10] K. P. Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux J.*, 1996(29es):7, 1996.

[11] C. J. Mauer, M. D. Hill, and D. A. Wood. Full-system timing-first simulation. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 108–116, 2002.

[12] E. Schnarr and J. R. Larus. Fast out-of-order processor simulation using memoization. In *Proceedings of the Eight International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 283–294, Oct. 1998.

[13] SimNow webpage. http://developer.amd.com/simnow.aspx.

[14] J. Smith and A. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th International Symposium on Computer Architecture*, Jan. 1985.

[15] E. L. T. Austin and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, Feb. 2002.

[16] M. T. Yourst. PTLSim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *Proceedings of ISPASS*, Jan. 2007.