Designing Hardware that Supports Cycle-Accurate Deterministic Replay

Brian Greskamp, Smruti R. Sarangi, and Josep Torrellas
Department of Computer Science, University of Illinois
http://iacoma.cs.uiuc.edu

Abstract

Most computer hardware today is nondeterministic, *meaning that two executions of a program will not be cycle-for-cycle identical at the microarchitectural level even if they start from the same microarchitectural state. Due to uninitialized state elements, I/O, and timing variations on high-speed buses, the microarchitectural states of the two executions will evolve differently.*

Such nondeterminism complicates system verification and makes hardware faults detected during bringup more difficult to reproduce and analyze. Consequently, we believe that board-level computer hardware should be designed in a way that supports cycle-accurate deterministic replay. In this paper, we outline the hardware required to provide this capability. We argue that the resulting hardware complexity is minimal, providing a net savings in bringup time and cost. We also show that potential applications of deterministic hardware extend far beyond hardware verification.

1. Introduction and Motivation

We propose the *Cycle-Accurate Deterministic REplay* (CADRE) architecture, which cost-effectively makes a board-level computer cycle-deterministic — including processors, buses, memory, chipset, and I/O devices. CADRE uses checkpoints, logs, and certain hardware extensions to enable replayed executions that match the microarchitectural state of the original execution cycle-for-cycle. For example, assume that one of the processors in the computer observes a bus signal transition A at internal cycle a and initiates an ALU operation B at cycle b. These events will recur at exactly the same internal cycles during the re-execution. Further, the microarchitectural states of the multiple processors, memory controllers, and other components will evolve exactly as they did during the original execution.

Cycle-accurate determinism has many applications, but one of the most obvious is in system bringup — the verification phase when engineers begin running programs on first silicon. Since the real processor is so much faster than the simulators used in earlier verification phases, longer and more detailed tests, such as booting a full operating system, can finally be executed. The bringup tests quickly reveal many previously unknown bugs, which must be characterized. The characterization process typically begins with finding a way to reliably reproduce an error. The engineer can then employ "iterative debugging" — replaying the error and examining system state before and after to gain a full understanding of the problem. With typical hardware, finding a test that reliably reproduces the error is difficult or impossible, but with CADRE, it is trivial. With CADRE, an engineer can replay a failing test over and over, with the assurance that at each cycle, the signal and state transitions will exactly match those of the original execution. He can then stop the machine at different points and examine the internal state through a test access port or read out the complete system state at any point and transfer it to an RTL simulator for detailed analysis.

Deterministic hardware is also easier to test than nondeterministic hardware. Already, automatic testers are encountering problems with nondeterminism [5]. These testers operate by presenting test vectors at the chip's input pins and observing the response vectors on the output pins. In a nondeterministic system, response vectors may not arrive at the tester at the expected time, or even in the expected order. In extreme cases, the data in the response vectors could differ from the expected values. Cycle-accurate deterministic hardware does not present these problems.

CADRE is not just for verification and test; it can be deployed in the field, providing hardware vendors with a powerful tool to debug customer-site failures. After the customer identifies what he believes to be a hardware error, he could send the vendor a checkpoint preceding the crash. The vendor would then be able to reproduce the fault exactly using in-house hardware and simulators. The idea is similar to the current use of software crash feedback agents that help software developers identify bugs in deployed software.

Cycle determinism also has less obvious applications. For example, a cycle-deterministic system is easier to incorporate into an n-way modular redundancy system. Traditional NMR systems, such as HP's NonStop server [2], require custom buffering and synchronization logic between the processors and voters because each processor may slowly slip behind or ahead of the others. Cycle-deterministic components do not require such compensation logic in NMR configurations, as cycle determinism ensures that as long as the components have the same inputs, they will continue in lock step.

Finally, hardware deterministic replay subsumes previous proposals for software determinism to debug parallel programs [11]. In a cycle-accurate deterministic system, the interleaving of replayed memory accesses is guaranteed to match the original, since all access occur at exactly the same cycle as during the original execution. Furthermore, as we will show later, interrupts and I/O events will also recur exactly where they should. One issue with this approach is that special measures are needed to allow a debugger to run on the target machine without interfering with hardware determinism during replay. A solution to this problem is to run the debugger on another machine attached to the target's front side bus or test access port.

2. Sources of Nondeterminism

A system supporting deterministic replay must have two key properties: (1) A deterministic execution interval must begin at cycle 0 with each state-holding element initialized to a known state. (2) A component must receive a signal at the nth edge of its local clock during replay iff it received the same signal at the nth edge during the original execution. The first condition is the base case, requiring that the original and replay executions start at exactly the same state, and the second is the inductive step, ensuring that they experience the same state transitions at the same cycles.

All nondeterminism is then traceable to one of two causes: (1) incomplete initialization, in which some state-holding elements are in an incorrect or unknown state at the start of replay, or (2) changes in the arrival times of signals, possibly due to environmental factors. Below, we discuss the two causes as they apply to each component of the system.

CPUs Modern processors contain millions of bits of state contained in registers, pipeline latches, SRAMs, and counters. Some state bits, like those in the branch predictor tables, have no architecturally-visible effect. Consequently, processors usually do not provide any ISA-level means of resetting them, violating condition 1. Additionally, dynamic power and temperature management techniques such as clock duty cycle modulation and voltage-frequency scaling (DVFS) are dependent on the environment (die temperature). As a result, the timing of power and temperature events is uncertain and condition 2 is violated.

Memory Systems The memory controller is the component responsible for scheduling memory read, write, refresh, and scrubbing operations. The Itanium-2 verification engineers [4] reported that the memory refresh and scrubbing operations are a source of nondeterminism because the scrubber and refresh walkers will be working on different lines during the re-execution than in the original. Therefore, scrubs and refreshes line up with the program's read and write accesses differently during replay. Due to contention, the timing of all memory operations will change.

I/O and Interrupts The timing of I/O operations and interrupts is notoriously unpredictable. For example, hard disks have mechanical components that introduce non-deterministic seek times and rotational delays. The timing of events from human-interface devices and network interfaces is equally unpredictable.

Buses The buses that cross clock domains in a computer, for example as they connect different chips, are a major source of nondeterminism. These buses are often source-synchronous [1], which means that the transmitter generates and transmits a clock signal that travels with the data to the receiver. One popular example is HyperTransport [3]. In these buses, receiving a message occurs in two steps (Figure 1). First, the rising edge of the transmitter clock signal latches the data into a holding queue in the bus interface of the receiver. We refer to this event as the *arrival* of the message. Some time later, normally on the next rising edge of the receiver's core clock, the receiver removes the data from the queue and submits it for processing. We refer to this event as the *processing* of the message.

Unfortunately, the exact arrival time is nondeterministic because all clock and data pulses that traverse the bus are affected by physical and electrical processes such as temperature variations, voltage variations, channel cross talk, and inter-symbol interference [1, 3, 8]. As a result, these signals experience a random but bounded delay on the bus and could arrive at any time during a certain window. For example, the HyperTransport specification assumes an uncertainty interval of one cycle even for very short buses [3]. Clearly, this is a violation of condition 2.

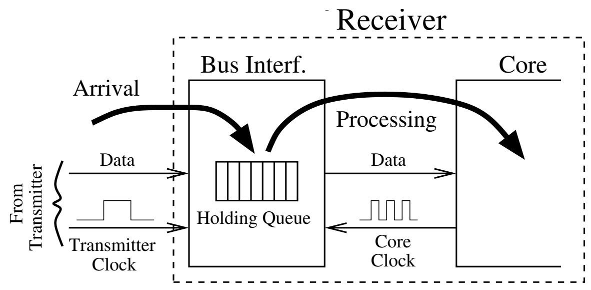

Figure 1. Arrival and processing of a message at the receiver.

Figure 2 illustrates how uncertainty in the arrival time of the transmitter clock can give rise to nondeterminism at the receiver. The receiver may see the rising edge of the transmitter clock arrive anywhere in the hatched interval. If the receiver processes the message on the first rising edge of the core clock after arrival, then the processing time is nondeterministic because it depends on the arrival time.

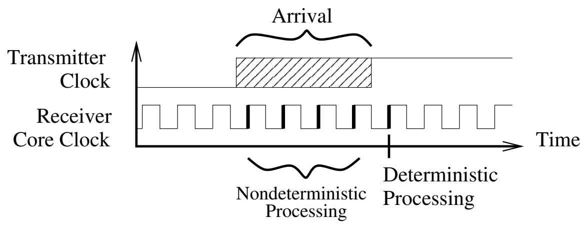

Figure 2. Nondeterministic and deterministic processing.

3. Ensuring Cycle Determinism in Buses

To make bus transfers fully cycle-deterministic, we propose to delay the *processing* of a message at the receiver until the last possible core clock cycle at which the message could have arrived. The correct processing time is shown in Figure 2 as "deterministic processing". The cost of this approach is a small increase in latency for some messages.

Our scheme works in any system where the ratio of the frequencies in the transmitter T and receiver R is constant, although the relative phase of the clocks may change with time (within bounds) due to physical and electrical effects. However, for simplicity, this paper will assume that T and R operate at the same frequency.

CADRE adds a *domain-clock* counter — an up-counter driven by the local clock signal — to both the transmitter and the receiver. At periodic global machine checkpoints (once per second), a broadcast signal resets all domain-clock counters. At any time, the difference between the transmitter's and receiver's domain-clock counts is bounded by $[p, q]$. Additionally, the transmission delay of a message on the bus (measured in domain-clock counts) is bounded by $[d_1, d_2]$. The constants d_1, d_2, p, and q are known at design time. As a result, if the transmitter sends a message at count x_T of its domain-clock counter, the message will *arrive* at the receiver at count y_R of the receiver's domain-clock counter, as given by:

$$y_R = x_T + [d_1 + p, d_2 + q] = x_T + [\theta_1, \theta_2] \quad (1)$$

We call $\theta_2 - \theta_1$ the *Uncertainty Interval*.

Our scheme to enforce bus determinism is detailed in [7]. It requires that the transmitter include in every message a short tag ρ (typically 1 or 2 bits) that allows the receiver to determine when the message was sent (x_T). Then, the receiver simply computes $z_R = x_T + \theta_2$ and delays the *processing* of the message until z_R, ensuring determinism.

The hardware required is shown in Figure 3. Our scheme adds a *Synchronizer* module to the bus interface of the receiver. This hardware processes the tag ρ arriving with the data, and uses it to determine at what cycle z_R to process the data. Full details are in [7].

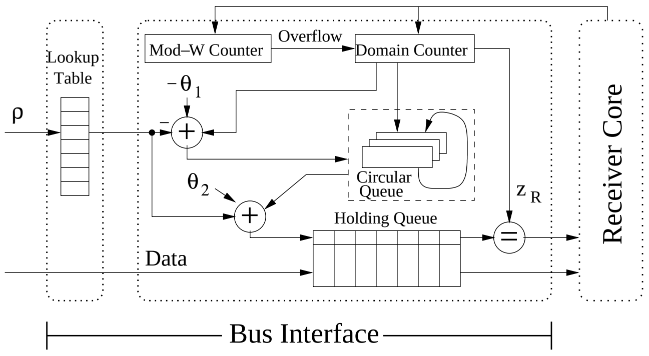

Figure 3. Synchronizer module added to the bus interface of the receiver.

4. Overall Deterministic System

To design a CADRE system, we build on the checkpointing and logging mechanisms from ReVive [6] or SafetyNet [10]. The idea is as follows. Periodically — say, once per second — processors write back their caches to memory and invalidate caches and TLBs. They then save their registers and completely re-initialize all internal state-holding elements to a known state. As execution proceeds after the checkpoint, when a main-memory location is about to be over-written for the first time since the checkpoint, the old value of that location is saved in a Memory Log. This is done in hardware by the memory controller. As discussed in [6, 10], this support enables memory state rollback.

To make each CPU deterministic, we introduce the `DETRST` instruction, which initializes all the state elements in the processor. `DETRST` is executed after every checkpoint. Moreover, each CPU is augmented with a CPU Log that records a variety of events, such as (i) clock duty cycle modulation, (ii) voltage-frequency scaling, and (iii) nondeterministic interrupts and exceptions generated inside the processor chip. Examples of the latter are thermal emergencies and ECC failures due to soft errors. During re-execution, events in the log are replayed to reproduce the events in the original execution.

To make the memory deterministic, we make two changes to the memory controller. First, the controller makes memory refresh deterministic by resetting the refresh logic at each checkpoint. In this case, if all of the inputs to the memory controller are deterministic, the refresh logic will generate deterministic outputs. As long as the checkpoint interval is long enough to allow at least one refresh operation to complete per memory location, the DRAM will not lose data. Moreover, to circumvent nondeterminism from memory scrubbing, the controller includes in the checkpoint the register that indexes the currently scrubbed line. When restoring the checkpoint, the register is restored, enabling scrubbing to resume from exactly where it was in the original execution.

Since I/O devices are inherently nondeterministic, CADRE uses a logging-based solution. Specifically, CADRE places a buffering module in the memory controller called the Input Log. The Input Log records all the messages arriving from the I/O devices and the interrupts that the I/O devices deliver. When replaying an execution, the I/O devices can simply be suspended by gating their clock and disconnecting them temporarily from the data bus. The Input Log will reproduce all of the signals that the I/O devices generated during the original execution.

Finally, to enforce determinism in source-synchronous buses, we use the module shown in Figure 3 at the receiver side of each bus. If a bus is bidirectional, CADRE places one such module at each end of the bus.

5. Feasibility

Possible concerns about CADRE include the added chip area, design complexity, storage overhead, and performance overhead. The area overhead of the added CADRE logic is very small; the bus synchronizers comprise fewer than a thousand gates each, and the memory and IO log controllers are also tiny. Only the Input Log, which is implemented in SRAM, consumes significant space. As for complexity, we feel that any additional design effort for a CADRE system is more than offset by the improvements in verification efficiency that cycle-accurate determinism provides.

The storage overhead is composed of the Input Log, CPU Logs, and the ReVive/SafetyNet Memory Log. The latter is estimated to be around 50 MB/s per processor in [11]. Although the size of the Input Log varies with the application, we found it to be quite low for a set of workloads that include SPECint, SPECfp, SPEComp, SPECjbb, and SPECweb. While some applications may require up to 100 MB/s during periods of high activity, no application exceeded 1 MB/s of input log bandwidth in the steady state. Finally, the storage cost of the CPU Log is negligible since frequency scaling and thermal events are rare in current processors.

The main contributor to performance overhead is the increased memory latency introduced by the bus synchronizers that make the path from the processor to memory deterministic. Other costs, such as flushing caches at checkpoints, are negligible with checkpoint intervals of one second or longer. So, assume that the memory controller is on a different chip than the processor and that the interconnection of the processor, memory controller, and memory modules is through HyperTransport links. Given current technology [3], the bus synchronizer on each link will add one cycle to each message in the worst case. We therefore consider a worst case of four additional bus cycles for each memory access. Our simulation results show that the resulting slowdowns on SPECjbb, SPEComp, SPEC-cpu, and SPECweb are all less than 1%.

To summarize, extending a four-way CMP server with CADRE hardware supporting a one second checkpoint interval would have a storage cost of about 200 MB of DRAM plus a few MB of SRAM (for the Input Log), a performance overhead of 1%, and a small area cost. For that price, we obtain the ability to "rewind" execution to a checkpoint one second in the past and re-execute deterministically cycle-for-cycle.

6. Related Work

The state of the art in deterministic replay for hardware debugging is Golan, a hardware testbed used to debug the Pentium-M processor [9]. Golan attaches a logic analyzer to the pins of the processor chip. Every input signal arriving at the pins is logged. This includes data to satisfy cache misses. Like CADRE, Golan takes a periodic checkpoint, which involves invalidating caches and TLBs, saving the processor registers, and resetting the processor state. Upon detection of a failure, Golan restores a checkpoint and restarts execution while replaying the logic analyzer log.

A shortcoming of the Golan approach is that the checkpoint interval (and therefore replay distance) is much shorter and storage overhead is much greater than in CADRE. Additionally, the probes that attach the logic analyzer to the processor pins present a difficult electrical design problem because the pins cycle at high frequency and no nondeterminism in the connection can be tolerated. Although Golan was highly successful in speeding Pentium-M bringup, it is not suitable for field deployment.

7. Conclusions

We have proposed that hardware that enforces cycle-accurate determinism be included in commodity computer systems to ease verification and testing, and for other purposes. We have explained what the main sources of nondeterminism are and how they can be circumvented. Finally, we have argued that the area, complexity, storage, and performance cost of the required hardware is minimal.

References

[1] W. J. Dally and J. W. Poulton. *Digital Systems Engineering*. Cambridge University Press, 1998.

[2] D. Bernick et al. NonStop advanced architecture. In *DSN*, pages 12–21, 2005.

[3] HyperTransport Technology Consortium. HyperTransport I/O link specification revision 2.00b, 2005.

[4] D. D. Josephson, S. Poehhnan, and V. Govan. Debug methodology for the McKinley processor. In *ITC*, pages 451–460, 2001.

[5] K. Mohanram and N. A. Touba. Eliminating non-determinism during test of high-speed source synchronous differential buses. In *VTS*, pages 121–127, 2003.

[6] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *ISCA*, pages 111–122, 2002.

[7] S. R. Sarangi, B. Greskamp, and J. Torrellas. CADRE: Cycle-accurate deterministic replay for hardware debugging. In *DSN*, 2006.

[8] L. Sartori and B. G. West. The path to one-picosecond accuracy. In *ITC*, pages 619–627, 2000.

[9] I. Silas et al. System level validation of the Intel Pentium-M processor. *Intel Technology Journal*, 7(2), May 2003.

[10] D. Sorin, M. Martin, M. Hill, and D. Wood. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *ISCA*, pages 123–134, 2002.

[11] M. Xu, R. Bodík, and M. D. Hill. A "Flight Data Recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA*, 2003.