CADRE: Cycle-Accurate Deterministic Replay for Hardware Debugging[*]

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

Abstract

One of the main reasons for the difficulty of hardware verification is that hardware platforms are typically nondeterministic at clock-cycle granularity. Uninitialized state elements, I/O, and timing variations on high-speed buses all introduce nondeterminism that causes different behavior on different runs starting from the same initial state. To improve our ability to debug hardware, we would like to completely eliminate nondeterminism.

This paper introduces the Cycle-Accurate Deterministic REplay *(CADRE) architecture, which cost-effectively makes a board-level computer cycle-accurate deterministic. We characterize the sources of nondeterminism in computers and show how to address them. In particular, we introduce a novel scheme to ensure deterministic communication on source-synchronous buses that cross clock-domain boundaries. Experiments show that CADRE on a 4-way multiprocessor server enables cycle-accurate deterministic execution of one-second intervals with modest buffering requirements (around 200MB) and minimal performance loss (around 1%). Moreover, CADRE has modest hardware requirements.*

1. Introduction

The complexity of computer hardware continues to grow. In current processor designs, verification typically consumes 50-70% of the design effort [4]. As a result, verification progress often determines the product development schedule. Meanwhile, ITRS [10] predicts a continued increase in design complexity and resulting verification effort. Consequently, it is crucial to develop techniques that ease hardware verification and debugging.

Like software debugging, the process of removing a hardware bug typically involves several loops of "iterative debugging". First, a bug is detected. Then, in each iteration, the hardware is returned to some state preceding the fault and the faulting sequence is re-executed. By monitoring internal signals and states during the re-executions, the designer gains an understanding of the bug.

Iterative debugging is most effective when the platform being debugged supports *cycle-accurate deterministic* execution. With deterministic execution, it is possible to replay the original faulting execution cycle-by-cycle. As long as re-execution starts from the same initial state as the original execution and is supplied with the same inputs at the same cycles, all the events of interest will re-occur at exactly the same cycles.

Cycle-accurate deterministic execution is easy to support in RTL simulation environments. However, RTL simulations are slow — about 100 cycles per second for a large design. As a result, hardware designers often have to run tests on real hardware at native speed. Unfortunately, board-level computer hardware does not ordinarily support cycle-accurate deterministic execution, which makes debugging much more difficult.

In this paper, we introduce 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. To design CADRE, we identify the main sources of nondeterminism in a board-level computer. Then, we present a novel scheme that circumvents one of the most difficult sources of nondeterminism, namely the timing of messages on source-synchronous buses that cross clock-domain boundaries. The proposed solution completely hides this nondeterminism at the cost of a small latency penalty and modest hardware. We construct CADRE by composing this scheme with other determinism and checkpointing techniques, many of which are already known.

We show that CADRE facilitates large windows of deterministic execution (one second or more) with modest storage overhead and negligible performance loss. In particular, experiments indicate that extending a four-way multiprocessor server with CADRE enables cycle-accurate deterministic execution of one-second intervals with buffering requirements of around 200MB and performance loss of around 1%.

CADRE has modest hardware requirements compared to current cycle-deterministic schemes. Moreover, its long intervals of deterministic execution are a substantial improvement over the tens of milliseconds between checkpoints achieved by current schemes. Long intervals enable verification engineers to effectively use native re-execution to debug hardware problems. Finally, CADRE's cost-effectiveness may enable its use in systems in the field.

This paper is organized as follows: Section 2 describes the sources of nondeterminism, Section 3 presents our scheme for making buses deterministic, Section 4 presents the CADRE architecture, Section 5 evaluates it, and Section 6 presents related work.

2. Sources of Nondeterminism

Our reference system is a board-level computer composed of *components* connected by *buses*, where a component is defined to have a single clock domain. In this paper, we consider the following components: the processor chip (which includes several cores),

[*]This work was supported in part by the National Science Foundation under grants EIA-0072102, EIA-0103610, CHE-0121357, and CCR-0325603; DARPA under grant NBCH30390004; DOE under grant B347886; and gifts from IBM and Intel.

the memory controller, the main memory, and the I/O controller. The buses of interest are those that connect components in different clock domains. While a bus could connect several components, all of the buses of interest considered in this paper happen to be point-to-point.

In this environment, assume that each component has a counter driven from its local clock. An *input* to component M is deterministic if it is guaranteed to arrive at the same count value in M on every run that starts from the same initial state. Likewise, an *output* of M is deterministic if it is always generated at the same count value in M. A *component* is deterministic if determinism of all its inputs implies determinism of all its outputs. Finally, a *system* is deterministic if all of its components and buses are deterministic.

This section enumerates the main sources of nondeterminism in CPUs, memory systems, IO systems, and buses. It also describes other effects.

2.1. Nondeterminism in CPUs

The traditional sources of nondeterminism in CPUs are the result of how the logic is designed. The principal sources of logic nondeterminism are: (i) random replacement policies in caches, (ii) random request selection in arbiters, and (iii) uninitialized state elements (i.e., elements in the Verilog ‘X’ state). In general, these sources can be eliminated with appropriate design of RTL code, for example as described by Bening and Foster [5].

Novel dynamic techniques for power saving such as clock duty cycle modulation and voltage-frequency scaling (DVFS) also contribute to nondeterminism. They affect the timing of events in the core. On the other hand, dynamic clock gating can be designed to maintain determinism. For example, the Pentium 4 is designed such that a hardware unit’s outputs are cycle-for-cycle equivalent irrespective of whether the unit is clock gated or not [6].

2.2. Nondeterminism in Memory Systems

The cores of synchronous DRAM memory chips have fixed latencies for the operations and, therefore, can be considered fully deterministic. This is reasonable because the control logic in the DRAM chip is relatively simple. However, there is more to the memory system than just the DRAM chips; a good deal of the memory’s “intelligence” is located in the memory controller. The memory controller is typically either a part of the chipset or is integrated with the processor core. It is responsible for scheduling memory requests and managing DRAM refresh and ECC scrubbing operations.

Of these operations, DRAM refresh and ECC scrubbing are sources of nondeterminism. This was pointed out by verification engineers working on the Intel Itanium-2 processor [13]. The reason is that refresh is typically implemented as a task that runs opportunistically depending on the memory conditions [3]. Therefore, as we re-execute an application, the timing of refreshes changes. The ECC scrubbing task in the memory controller contributes to nondeterminism for the same reason.

2.3. Nondeterminism in IO and Interrupts

The timing of IO 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.

2.4. Nondeterminism in Buses

The buses that cross clock domains in a computer, for example as they connect different chips together, are a major source of nondeterminism. These buses are often source-synchronous [8], which means that the transmitter generates and transmits a clock signal that travels with the data to the receiver. One popular example is HyperTransport [11]. 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 them for processing. We refer to this event as the *processing* of the message.

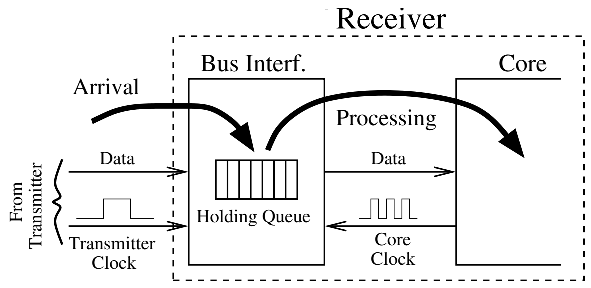

Figure 1. Arrival and processing of a message at the receiver.

The causes of nondeterminism are *jitter* and *drift*, which change the arrival times of clock and data pulses at the receiver. They are the inevitable result of physical and electrical processes such as temperature variations, voltage variations, accumulated phase error in PLLs, channel cross talk, and inter-symbol interference [8, 11, 17]. All high-speed communication specifications must therefore include provisions for tolerating the uncertainty in signal arrival times, but they do not usually guarantee determinism.

Figure 2 illustrates how uncertainty in the arrival time of the transmitter clock can give rise to nondeterminism at the receiver. Due to drift and jitter, the receiver may see the rising edge of the transmitter clock 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.

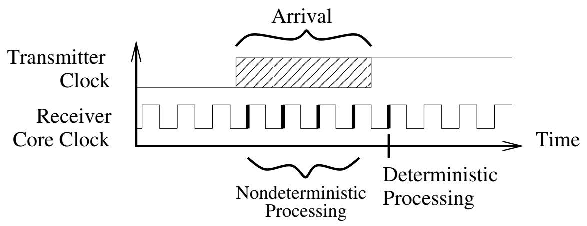

Figure 2. Nondeterministic and deterministic message processing.

Current systems experience nondeterminism. For example, the HyperTransport protocol assumes a sampling error of one cycle even for very short buses [11].

Even high-precision chip testers will find it difficult to maintain determinism as design frequencies increase. The arrival time uncertainty for testers is expressed in terms of the EPA (Edge Placement Accuracy), which is defined as the 3σ variation in the difference between the actual and intended arrival times of a signal transition at the receiver [15]. Commercial test equipment typically has a 25 ps EPA [19], but even with such high accuracy, the probability of nondeterminism over a long run is substantial. Figure 3 shows, for a range of EPAs, the probability that a receiver with a given core clock frequency processes at least one message in the wrong clock cycle, as it re-executes a run of one million messages. Even with a tester-quality 25 ps EPA, designs clocked at 5 GHz or higher are likely to experience nondeterminism.

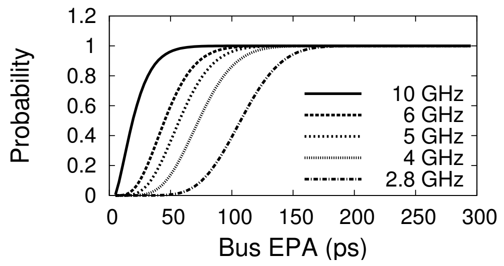

Figure 3. Probability of nondeterminism after one million messages as a function of the EPA for different receiver core clock frequencies.

2.5. Other Issues: Circuit Faults

Another source of nondeterminism is circuit faults. Early silicon for modern high-speed processors can have difficult-to-diagnose signal integrity problems, which can manifest as intermittent failures. The CADRE architecture does not guarantee reproducibility of circuit-level electrical faults. However, by making the other aspects of the system cycle-deterministic, CADRE can create similar electrical conditions during the re-execution and, therefore, *help* reproduce these faults.

Similarly, a Single Event Upset (SEU) fault due to a particle strike can prevent CADRE from re-executing cycle-deterministically. However, if we can detect SEU faults, it could be possible to record them during the original execution and mimic them during re-execution. These issues are beyond the scope of this paper.

3. Enforcing Determinism in Buses

In a computer's source-synchronous buses, the transmitter and receiver clocks typically have a bounded skew. Specifically, since the two clocks are typically derived from a common reference clock, the ratio of their frequencies is constant. However, their relative phase changes with time (within bounds) due to physical and electrical effects[1]. For these very common bus scenarios, we propose an approach to make the bus transfer fully deterministic.

To understand the approach, consider Figure 1. The idea is 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 very small increase in the communication latency for many of the messages sent.

To determine the delay that we should enforce, consider a system where the transmitter T and receiver R run at frequencies f_T and f_R, respectively. Both transmitter and receiver have a *domain-clock* counter, which is an up-counter driven by the local clock signal. Such counters are reset with a global signal at machine checkpoints — in our case, at roughly every second.

Let us first consider the case when $f_T = f_R$. Let n_T and n_R be the transmitter's and receiver's domain-clock counts, respectively, at a given time. Such counts can differ by a bounded value. Let us assume that their difference is $n_R - n_T \in [p, q]$. Let us also assume that the transmission delay of a message in the bus (measured in domain-clock counts) is bounded by $[d_1, d_2]$. The values of p, q, d_1 and d_2 are known. Now assume that the transmitter sends a message at count x_T of its domain-clock counter. At the same time, the receiver's domain-clock count is x_R. The message *arrives* at the receiver at count y_R of its domain-clock counter. From the previous discussion, y_R is:

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

Figure 4(a) shows a receiver timeline and two possible transmitter timelines: one is the case with the smallest possible count difference between y_R and x_T, and the other is the case with the largest. We call $\theta_2 - \theta_1$ the *Uncertainty Interval*.

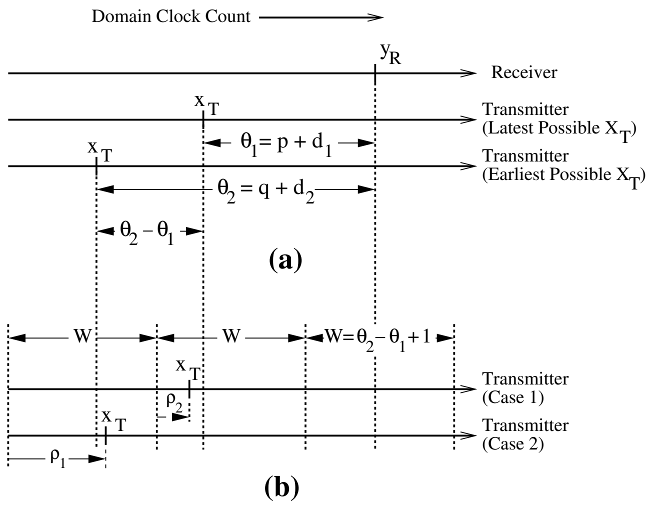

Figure 4. Timing of transmitter and receiver events.

To ensure deterministic timing, our approach requires the transmitter to send some count information to the receiver. With this information, the receiver can determine when the message was sent (x_T). Then, the receiver simply computes $x_T + \theta_2$ and delays the

[1]These clocks where the relative frequency does not change but the relative phase may change within bounds are called mesochronous [8].

processing of the message until that point, therefore ensuring determinism. Consequently, the Uncertainty Interval is the maximum delay that we need to add to a message to ensure bus determinism.

In the following, we present two ways to support our approach. The first one involves sending count information with every message; the second one sends count information at every cycle. Later, we consider the case of $f_T \neq f_R$ and present a hardware implementation.

3.1. Sending Information with Every Message

A simple approach is for the transmitter to attach the current value of its domain-clock count (x_T) to every message. This approach, called *FullCount*, requires adding many bits to each message. For instance, if the interval between checkpoints is 1 second and f_R is 1GHz, then the count takes at least 30 bits. For narrow buses, this adds multiple cycles to each message. Still, for long messages like memory lines, adding four additional bytes makes little difference.

An alternative approach called *Offset* is to divide the time into windows of a fixed number of cycles. The transmitter attaches to every message only the current *offset* count (ρ) from the beginning of the current window (Figure 4(b)). For this approach to work, the *Window Size* W must be such that, given ρ, the receiver is able to reconstruct without ambiguity the window number (N_W) where x_T is found. Then, on reception of ρ, the receiver can reconstruct x_T as $N_W \times W + \rho$.

In practice, any window size larger than the Uncertainty Interval will do:

$$W = \theta_2 - \theta_1 + k \qquad (2)$$

where k is an integer larger than zero. In our initial design, we use $k = 1$. To see why this works, consider Figure 4(b). The figure shows two examples of transmission times x_T that are in between the earliest and the latest possible values. Any x_T has an associated $\rho = x_T$ mod W that gives the offset from the beginning of its window. Since the Uncertainty Interval is smaller than W, it is either fully inside a window or strides two windows. In the latter case, the values of ρ in the earlier window are strictly greater than the values of ρ in the later window. Consequently, in all cases, $y_R - \theta_1 - \rho$ lies in the same window as x_T. This means that $\lfloor (y_R - \theta_1 - \rho)/W \rfloor$ is x_T's window number N_W. Given N_W, the receiver reconstructs x_T as $N_W \times W + \rho$. The receiver then processes the message at count $z_R = x_T + \theta_2$.

This approach has the advantage that ρ needs very few bits. For example, as per the HyperTransport protocol for very short buses [11], we can assume an Uncertainty Interval equal to one. Then, we can set W to two and, since ρ only needs $\log_2 W$ bits, we only need one bit for ρ. Even if we assume an Uncertainty Interval of three cycles, we only need two bits for ρ. In this case, we can either add two extra bits to the bus or consume only one extra cycle per message even for very narrow buses.

3.2. Sending Information At Every Cycle

In a different approach called *Pulsing*, the transmitter clock cycles continuously. The receiver has one additional counter that is incremented at each pulse of the transmitter clock. With this information, the receiver identifies each cycle. When a message arrives at the receiver, the receiver is able to determine at what cycle count x_T the transmitter sent the message. Knowing x_T and its current domain-clock count, the receiver can delay message processing appropriately to ensure determinism. At each checkpoint, the pulses temporarily stop while the counter is being reset.

This scheme requires a dedicated *Idle* signal. If the transmitter has no data to send, it cannot stop sending the clock pulses. Instead, it asserts the Idle signal and continues clocking. Overall, this scheme is very inexpensive for point-to-point unidirectional buses.

However, consider a bus connecting several components. In the schemes of Section 3.1, all components send their x_T or ρ over the same wires — we size the number of wires for ρ based on the largest W of all the components. Unfortunately, for *Pulsing*, we need to allocate one pair of dedicated clock and Idle wires for each component. The wires cannot be shared because the clock lines are busy all the time.

3.3. Different Transmitter & Receiver Frequencies

We address the case when the transmitter frequency f_T and the receiver frequency f_R are different by assuming a hypothetical transmitter T' associated with the true transmitter T (Figure 5). T' cycles at the same frequency as the receiver and is perfectly synchronized with T — there is no signal drift or jitter in their communication.

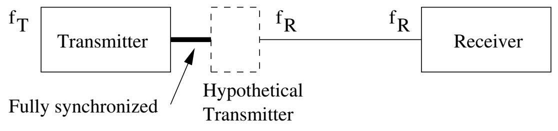

Figure 5. Adding a hypothetical transmitter when transmitter and receiver frequencies differ.

With this approach, any cycle counts in T are scaled to cycle counts in T' using the ratio of frequencies:

$$n_{T'} = \lceil n_T \times f_R / f_T \rceil \qquad (3)$$

Then, we take the analysis developed so far and reuse it for the communication between T' and R, which cycle at the same frequency. The values of all the input parameters (θ_1, θ_2, and W) and ρ are given in f_R cycles.

We can use any of the schemes of Sections 3.1 and 3.2. For example, if we use the *FullCount* scheme, T' hypothetically attaches $\lceil x_T \times f_R / f_T \rceil$ to every message.

If, instead, we use the *Offset* or *Pulsing* schemes, we need a translation table that maps cycle counts in f_T to cycle counts in f_R. To see how this works, consider the Window Size, whose value in ns is constant but whose value in T' and T cycles is different — $W_{T'}$ and W_T, respectively. To select the Window Size, we need to find $W_{T'}$ and W_T that are both integers and are related as per $W_{T'} = W_T \times f_R / f_T$. For this, we may need to select a k in Equation 2 that is greater than one. Note that the values θ_1, θ_2, and k in Equation 2 are cycle counts and, therefore, are different integer values in f_T and f_R cycles. After we find $W_{T'}$ and W_T, we use a lookup table with as many entries as W_T. In each entry i, we store $j = \lceil i \times f_R / f_T \rceil$. This table maps cycle counts in the two frequencies. As an example, assume that $f_T = 300$ and $f_R = 500$, and that we end up setting $W_T = 6$, and $W_{T'} = 10$. The six entries in the table contain $[0, 2, 4, 5, 7, 9]$.

In *Offset*, we use this table to translate the ρ_T that T attaches to messages (in the example, 0, 1, 2, 3, 4, or 5) to the $\rho_{T'}$ that T' hypothetically attaches to the same messages (0, 2, 4, 5, 7, or 9). Conversely, in *Pulsing*, the table contains the only cycle counts in f_R at which T' hypothetically sends pulses.

This approach applies irrespective of which frequency is higher. The case of $f_R < f_T$ requires that the transmitter does not transmit all the time — otherwise, the receiver would overflow. It may result in multiple T cycle counts being mapped into the same T' cycle count. This does not cause nondeterminism, as long as the messages are queued and processed in order.

3.4. Hardware Implementation

Finally, we present an implementation of our approach. We focus on the *Offset* scheme for $f_T \neq f_R$, since this is the most advanced and general scheme.

As indicated in Section 3.1, the receiver computes the cycle at which to process a message with:

$$z_R = \lfloor (y_R - \theta_1 - \rho_R)/W \rfloor \times W + \rho_R + \theta_2 \qquad (4)$$

where all these parameters are given in f_R cycles. In particular, we use ρ_R to refer to the offset in f_R cycles.

Our design introduces logic in the bus interface of the receiver (Figure 6). The logic performs two operations, namely translating the ρ_T generated by the transmitter to ρ_R, and computing Equation 4. We perform the first operation as outlined in Section 3.3, using a small lookup table that is loaded at boot time. Placing the table in R (leftmost part of Figure 6) frees the T from having to know about R's frequency.

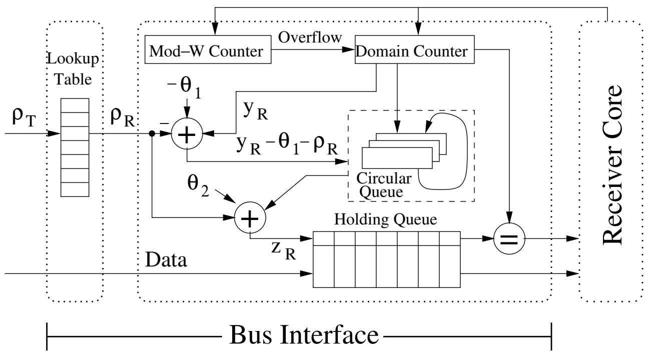

Figure 6. Implementation of the *Offset* scheme when transmitter and receiver frequencies differ.

To implement Equation 4, we use the rest of the hardware in Figure 6. First, we read y_R from the current value of the domain-clock counter and compute $m = y_R - \theta_1 - \rho_R$. Then, to compute $q = \lfloor m/W \rfloor \times W$, we save the value of the domain-clock counter periodically at every W cycles into a short circular queue. The saving operation is triggered by an overflow signal from a modulo-W counter. The q value is the highest value in the circular queue that is lower than m. This operation requires a CAM search on the queue. Fortunately, it can be shown that the number of queue entries is small, as it is bounded by $2 + \lceil \theta_1/W \rceil$. Finally, we add $q + \rho_R + \theta_2$ to obtain z_R.

The resulting z_R and the data coming from the bus are placed into the Holding Queue. At all times, the value of the domain-clock counter is compared to the z_R of the entry at the head of the queue. When both match, the entry at the head of the queue is processed.

4. CADRE Architecture

Figure 7 shows a computer augmented with the CADRE architecture and its determinism boundary. The example machine has the board-level architecture of a typical Intel-based system. The processor chip connects to a Memory Controller Hub (MCH) chip, which manages memory accesses, refreshes, and integrity checks. The MCH communicates with the IO Controller Hub (ICH), which controls peripheral buses such as ATA and PCI.

Our system uses a multiprocessor chip with a single clock domain. The cores are connected by a fully-synchronous on-chip bus and, therefore, core-to-core communications within the chip are assumed deterministic. However, the buses connecting the processor chip to the MCH, and the MCH to the memory modules and ICH are source-synchronous and, therefore, nondeterministic.

CADRE consists of a set of architectural modules that ensure deterministic execution. CADRE also leverages support for system checkpointing and rollback as proposed elsewhere [16, 18]. Figure 7 shows the CADRE modules in a shaded pattern. In the following, we describe the architecture in detail.

4.1. Support for Deterministic Execution

4.1.1. Deterministic CPUs

To make each CPU deterministic, we require a means of setting it to a known hardware state at every checkpoint. For that, we save the architectural registers, flush the pipeline, and write back and invalidate caches and TLBs. Then, we need to initialize all the counters, buffers, and other state elements in the CPU to a known state. Unfortunately, modern processors do not typically provide this capability. Indeed, Dahlgren *et al.* [7] observe that many flip-flops can have different contents across different runs because of non-deterministic initial values. Therefore, CADRE assumes the existence of an instruction, `DETRST`, that initializes all these state elements. The Pentium-M debugging platform [12] already provides the ability to reset certain state elements inside the processor. Thus, we believe that `DETRST` can be implemented in a production processor with modest performance cost, and should need five to ten cycles to reset all these processor structures [21].

CADRE augments the CPU with a *CPU Log* that logs a variety of events (Figure 7). These events are (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. Each entry in the log contains the value of the domain-clock counter and an event descriptor. During re-execution, events in the log are replayed to reproduce the events in the original execution.

Handling frequency scaling events requires additional steps. This is because, when the frequency changes, many events in the processor may become nondeterministic. In addition, the module for bus determinism discussed in Section 3.4 needs to be informed. Consequently, when the frequency needs to be changed, CADRE updates the CPU Log and forces a checkpoint. After that, it changes

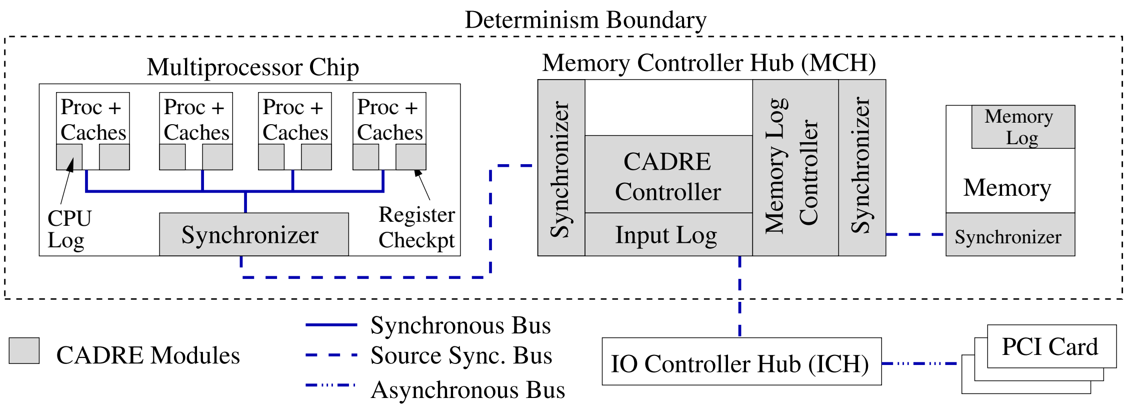

Figure 7. Computer augmented with CADRE modules.

the frequency, informs the bus, and then restarts execution. This scheme enables a deterministic replay.

4.1.2. Deterministic Memory System

A *CADRE Controller* in the MCH ensures determinism in the memory system (Figure 7). 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 location, the DRAM will not lose data.

To circumvent nondeterminism from memory scrubbing, the CADRE Controller includes in the checkpoint the MCH register that indexes the currently scrubbed line. When restoring the checkpoint, the register is restored, enabling scrubbing to resume exactly from where it was before the checkpoint.

4.1.3. Deterministic IO

Since IO devices are inherently nondeterministic, CADRE uses a logging-based solution. Specifically, CADRE places a buffering module in the MCH called the *Input Log* (Figure 7). The Input Log records all the messages arriving from the ICH. In addition, it also captures the interrupts that the ICH delivers to the MCH and the non-deterministic interrupts that are generated in the MCH. Each entry of the Input Log records the event and the domain-clock count.

The Input Log obviates the need to checkpoint IO devices such as network cards, sound cards, disk controllers, and graphics cards connected to the ICH. When replaying an execution, the IO 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.

Unlike other IO devices, a graphics card is typically connected directly to the MCH. In this case, CADRE can still treat it as an IO device and therefore record inputs from it in the Input Log. Alternatively, if the graphics unit can be checkpointed, it can be treated as a deterministic agent like a processor.

In Intel chipsets, the bus that carries data between the ICH and the MCH is narrow — at most 16 bits [2]. Consequently, each entry of the Input Log can be 48 bits wide: 16 bits of data and 32 bits for the domain-clock timestamp. For space efficiency, rather than logging the full 32-bit timestamp, we can log only the offset from the previous logged timestamp.

4.1.4. Deterministic Buses

Section 3 proposed a mechanism to enforce determinism in source-synchronous buses. The mechanism requires the module shown in Figure 6 at the receiver side of a unidirectional bus. Since the buses in our reference machine are bidirectional, CADRE needs one such module at both ends of each bus. The module is shown as *Synchronizer* in Figure 7.

At the beginning of each checkpoint interval, the CADRE Controller in the MCH broadcasts a BUSRST signal that resets the domain-clock counters in all the Synchronizers. This signal is sent over the buses and may not reach all the Synchronizers at the same absolute time. This is not a correctness problem. Instead, as indicated in Section 3, it gives rise to the p and q terms of Equation 1, which increase the latency of deterministic buses.

4.2. Checkpointing and Replay

For CADRE's deterministic execution to be usable, CADRE also needs to be able to checkpoint, rollback, and replay execution. To do so, CADRE can use a simplified version of the hardware checkpointing mechanisms proposed for Revive [16] or SafetyNet [18]. Specifically, at periodic times (e.g., every second), CADRE checkpoints the machine by: (i) saving the architectural registers, processor state registers, and message queues; (ii) writing back and invalidating all caches and TLBs; and (iii) invalidating predictor tables. 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 Log Controller* in the MCH (Figure 7). As discussed in [16, 18], this support enables memory state rollback.

4.2.1. Checkpointing Protocol

CADRE's checkpointing protocol is a variation of the classic Chandy–Lamport algorithm for checkpointing distributed systems [14] (Figure 8(a)). The CADRE Controller in the MCH coordinates the two phases of the algorithm. In the first phase, it broadcasts a message to all the agents in the determinism boundary (typically the MCH and all the processors), asking them to stop

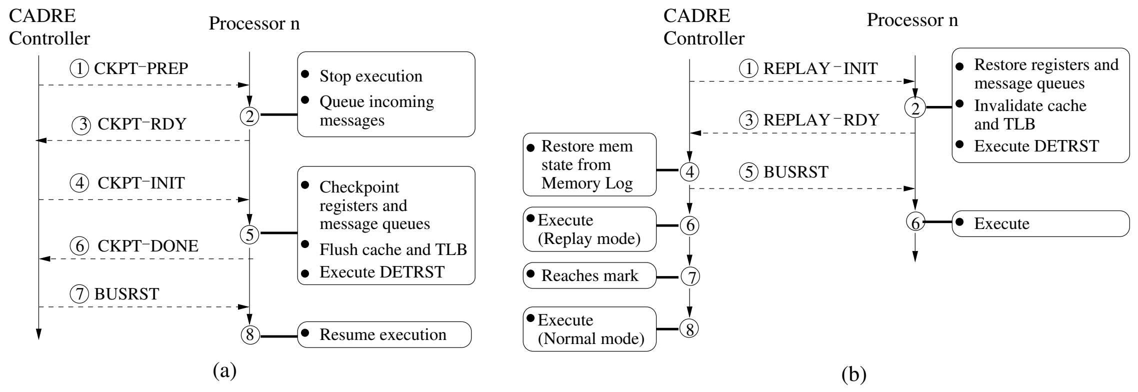

Figure 8. Checkpoint protocol (a) and replay protocol (b).

execution. In the second phase, the agents checkpoint their state and then reset it to a deterministic state. The detailed algorithm follows.

Checkpoint Initiation. When the maximum time between checkpoints elapses or one of the logs (CPU, Input, or Memory Log) becomes near full, the CADRE Controller initiates a new checkpoint by sending the CKPT-PREP message to all agents. Upon receiving CKPT-PREP, an agent suspends execution and sends a CKPT-RDY message back to the MCH.

Wait State. The agent remains stopped and any incoming messages are queued. Meanwhile, the CADRE Controller is waiting to receive CKPT-RDY responses from all agents. When it gets them all, the Controller sends CKPT-INIT to everyone.

Checkpoint State. When an agent has received CKPT-INIT on all incoming buses, it enters the checkpoint state. It saves the architectural and processor state registers and the message queues which contain data that has been accumulating since execution stopped. It also writes-back and invalidates caches and TLBs, and invalidates predictor tables. Finally, it executes the DETRST instruction to reset the hardware. At this stage, it is possible to transfer the execution to an RTL simulator by simply initializing the simulator state with DETRST and loading the checkpoint data.

When the agent finishes taking the checkpoint, it sends a CKPT-DONE response to the CADRE Controller. After receiving all the CKPT-DONE responses, the CADRE Controller asserts the BUSRST signal on all buses. In response, all agents reset their domain-clock counters to zero and resume execution.

4.2.2. Replay Protocol

The replay protocol has three phases, namely checkpoint restoration, deterministic replay, and return to normal execution (Figure 8(b)).

Checkpoint Restoration. The CADRE Controller first saves the current MCH domain-clock counter value in the *Replay Stop Marker* register. Then, it sends the REPLAY-INIT message to all agents. On message reception, agents restore the saved registers and message queues, invalidate (but *not* write-back) caches and TLBs, and execute DETRST. Each agent then freezes its pipeline and sends REPLAY-RDY to the MCH. On receiving REPLAY-RDY from all agents, the CADRE Controller reads the Memory Log and restores memory to the previous checkpoint. Finally, it asserts the BUSRST signal. Upon seeing the BUSRST signal, all agents reset their domain-clock counters and resume computing, therefore replaying the execution.

Deterministic Replay. The MCH behaves just as during normal execution except that IO inputs are replayed from the Input Log and IO outputs are discarded. IO devices are completely isolated from the system. A supervisor unit in each processor plays back the CPU Log, reproducing the clock duty cycle modulation events, voltage-frequency scaling events, and nondeterministic interrupts and exceptions that occurred during the original execution. We assume that no unexpected thermal emergencies occur during replay — since we are replaying the execution deterministically, the thermal profile of the re-execution follows that of the original one.

Return to Normal Execution. When the MCH domain-clock counter reaches the value stored in the Replay Stop Marker register, the deterministic replay is complete. CADRE switches back to Normal mode and the IO devices are reconnected.

5. Evaluation

Rather than evaluating CADRE with a traditional simulation approach, we estimate its performance and storage overheads through measurements on a real, non-CADRE system. A key benefit of this approach is the ability to run real programs to completion. A shortcoming is that some CADRE overheads are not modeled and that validation of the proposed algorithms and techniques is not as thorough.

We take an Intel server and estimate the main CADRE performance and storage overheads. Specifically, we measure the performance overhead of (i) periodic cache and TLB writeback and invalidation, and (ii) longer main memory latencies (which would be caused by the bus synchronizers). These are the main CADRE performance overheads. Less important CADRE performance overheads that are not measured include (i) the periodic invalidation of branch predictors and potentially similar structures, and (ii) the memory logging of first writes. Such logging has been shown to have negligible overhead in [16, 18] because it is not in the critical path of execution.

To estimate storage overheads, we measure the rate of IO input, which determines the size of CADRE's Input Log. To estimate the size of CADRE's Memory Log, Section 5.1 discusses the results reported in the literature for checkpointing schemes. Finally, the size of CADRE's CPU Log is negligible because it stores only rare events.

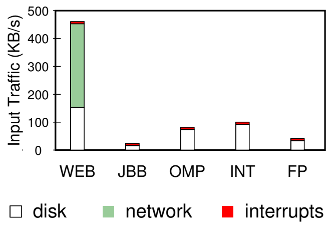

Figure 9. Mean input IO bandwidth.

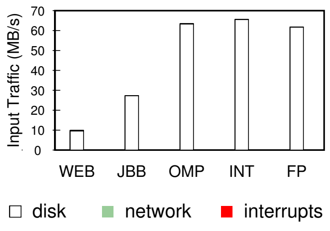

Figure 10. Peak input IO bandwidth.

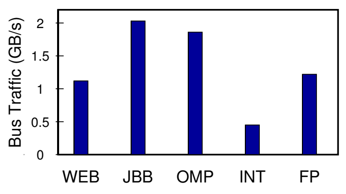

Figure 11. Mean input bus bandwidth.

Table 1 shows the characteristics of the server measured and the applications executed. SPECweb and SPECjbb use Sun's JDK 1.5. SPEComp is compiled with Intel's C and FORTRAN compilers with –O2. SPECint and SPECfp are compiled with gcc and f77 with –O3. SPECweb is excluded from the performance experiments because its response time varies greatly from run to run even on the base unmodified hardware — it is therefore difficult to separate estimated CADRE slowdowns from the noise. In our experiments, for each of the SPEComp, SPECint and SPECfp suites, we run all the applications in the suite in series and report the overall results. The *galgel* application from SPEComp is excluded from our experiments because it generates runtime errors on our machine.

Server Measured
Dual-processor 2.8 GHz Pentium-4 Xeon server with Hyperthreading (4 hardware threads total) Per-processor L2 cache: 1 MB Main memory: 2 GB of DDR2-400 SDRAM Chipset: Intel E7525 with 800 MHz front side bus OS: SuSE Linux 9.3 with 2.6.11 kernel
Workloads Executed
SPECweb2005: Enterprise webserver running Apache 2.0 with 500 clients. It performs intensive network and disk IO, but it is CPU bound. Used in the IO experiments only SPECjbb2005: Enterprise Java middleware with very little IO SPEComp2001: Parallel scientific applications in C or FORTRAN with OpenMP parallelization pragmas SPECint2000 and SPECfp2000: CPU-intensive sequential codes

Table 1. Experimental setup. In the plots, we label the workloads *WEB*, *JBB*, *OMP*, *INT*, and *FP*.

In the following, we first measure the space and performance overheads, and then address related issues.

5.1. Space Overhead

The two main storage overheads in CADRE are the Input Log and the Memory Log. To estimate the size of the former, we periodically poll the Linux kernel for disk, network card, and interrupt controller transfer rates as the workloads run. Recall that CADRE only needs to log *input* IO.

Figure 9 shows the average data rate for each of the three input IO sources and each workload. Figure 10 shows the peak rate, measured over 100 ms intervals. The figures show that the sustained input rate is quite low, even for SPECweb. The peak rate, however, is much higher. The highest rates are seen during application startup. From these figures, we conclude that if we wish to provide a one-second replay interval for these workloads during startup, we must buffer approximately 64MB of input IO data; for a one-second of steady state, the requirement for the workloads other than SPECweb is 100KB.

We cannot easily use our experimental setup to estimate the size of CADRE's Memory Log. Consequently, we report results from checkpointed systems in the literature that similarly log in memory on first write. Specifically, SafetyNet [18] logs on average 50MB/s per processor for the most memory-intensive workload (SPECjbb) [22]. The authors simulate 4GHz single-issue in-order processors. ReVive [16] generates a maximum log of 125MB/s per processor for the most memory-intensive workload (FFT) — and, on average for all the applications, a maximum log of 38MB/s. The authors simulate 1GHz 6-issue out-of-order processors.

These log rates are conservative for CADRE because they were measured for short checkpoint intervals (330ms and 10ms). First-write log storage overhead increases sublinearly as we increase the checkpoint interval to CADRE's one second. Moreover, logs can use compression to reduce their size to less than half [22]. Overall, if we assume a rough figure of 50MB/s per processor, our four-core machine would require 200MB to support a one-second checkpoint interval. While exact space overhead numbers will vary depending on workload, processor, memory, and IO system, both the Input and Memory Log in CADRE have tolerable sizes.

5.2. Performance Overhead

As indicated before, the two main performance overheads in CADRE are the periodic cache and TLB writebacks and invalidations, and the longer main memory latencies induced by the bus synchronizers. To estimate the first overhead, we develop a kernel module that periodically executes the *WBINVD* instruction on all processors. Such instruction forces the writeback and invalidation of all on-chip caches and TLBs. We vary the frequency of WBINVD execution. Our results show that, with one-second intervals between executions, the performance overhead is negligible for all workloads. Even for intervals of 100ms, the overhead is less than 1%. Consequently, for the one-second CADRE checkpoints, this operation has no performance impact.

Consider now the latencies due to the bus synchronizers. Section 3 showed that the maximum delay that the bus synchronizer adds to a message to ensure determinism is equal to the bus Uncertainty Interval ($\theta_2 - \theta_1$). As per the HyperTransport protocol for very short buses [11], we assume an Uncertainty Interval equal to one for our buses. In a round trip from processor to memory, there are four accesses to source-synchronous buses. With an MCH running at 800MHz, this amounts to adding, in the worst case, four 800MHz cycles or 14 processor cycles. Consequently, in the worst case, CADRE adds 14 processor cycles to a round trip that takes ≈ 200 cycles.

To estimate the impact of this additional latency, we program the MCH to add 14, 28, 42, or 56 processor cycles to the memory latency. We do this by increasing the programmable read pointer delay, the RAS–CAS delay, and the clock guard band in the MCH [3]. Figure 12 shows the resulting slowdown of the workloads for the different memory latency increases. The figure shows that the increased latency has a small but measurable effect. If we consider the most realistic latency increase, namely 14 cycles, we see that the overhead is 1% or less for all workloads.

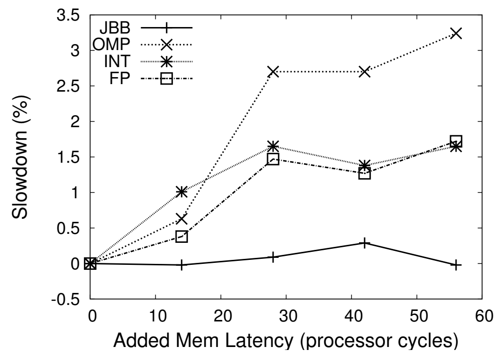

Figure 12. Slowdown due to longer latency to main memory.

5.3. Comparison to the State of the Art

We now compare CADRE to the state-of-the-art in cycle-deterministic systems. The state of the art is represented by Golan, a hardware testbed used to debug the Pentium-M processor [12]. Golan attaches a logic analyzer to the pins of the processor chip. Every input signal (address, data, and control) arriving at the pins is logged along with the clock cycle count. Periodically, Golan takes a checkpoint, which involves writing back and invalidating caches and TLBs, saving the processor registers, and performing a DETRST-like operation to reset the processor state. Upon detection of a failure, Golan restores a checkpoint and re-starts execution while replaying the logic analyzer log.

Although Golan also provides additional capabilities such as signal injection (useful for emulating I/O devices) and shmoo automation, we use it as an archetype of an approach to support cycle-determinism that we call *BusSnoop*. In BusSnoop, a buffering device snoops the pins of the processor chip and logs all the incoming events. Table 2 compares CADRE to *BusSnoop*.

Characteristic	*CADRE*	*BusSnoop*
Hardware Complexity	Moderate	High
Replay Distance	High	Low
Storage Needed	Low	High

Table 2. Comparing CADRE to the state of the art in cycle-deterministic systems.

We argue that CADRE substantially reduces the hardware complexity relative to BusSnoop. In BusSnoop, the buffering device interfaces to the high-frequency processor pins, which requires sophisticated signal tapping support. Alternatively, if the tapping occurs on the bus inside the chip, BusSnoop may require additional chip pins to dump on the fly the state that is being collected. Instead, CADRE buffers state at MCH/memory speeds (the Memory Log) or IO speeds (the Input Log), which is much simpler to implement. CADRE adds the CADRE Controller and bus synchronizers, which we believe have modest complexity. Finally, both CADRE and BusSnoop need similar hardware to ensure CPU determinism, which includes a mechanism like the CPU Log and appropriate RTL design of the CPU (Section 2.1). Overall, we feel that the reduction in hardware complexity is an important issue, especially as we suggest incorporating CADRE into systems deployed in the field (Section 5.4).

For a fixed storage requirement, CADRE is able to replay a much longer execution than BusSnoop. This is because BusSnoop needs to log much more information. Specifically, consider a 200MB log. As indicated in Section 5.1, this is our estimate of the size of the CADRE Memory Log needed to support one-second replays in our machine. To a first approximation, we neglect the Input Log requirements because, after program startup, they typically amount to noise over the Memory Log. On the other hand, BusSnoop fills 200MB very quickly. To see why, consider Figure 11, which shows the *input* data rates in the front side bus for each workload. These results are obtained with VTune [1]. These rates are typically 1-2 GB/s, with SPECjbb generating 2.03 GB/s. Let us assume that a BusSnoop log entry stores a 32-bit timestamp and a 64-bit data field. If we run SPECjbb, a 200MB buffer fills in only $(200\text{MB/s}/2.03\text{GB/s}) \times \frac{2}{3} \approx 66\text{ms}$. Consequently, BusSnoop's replay period is shorter.

Finally, from the previous discussion, we see that to support the same replay distance, CADRE needs much less log storage than BusSnoop. More specifically, BusSnoop's storage size is proportional to the duration of the period we want to replay. On the other hand, CADRE's Memory Log size increases only sublinearly with time because first-writes become less frequent with time. In fact, CADRE can support even an almost unbounded replay period with a checkpoint storage size equal to the memory size. In this case, we save the whole memory state to disk and run CADRE without Memory Log. CADRE then only needs to log IO. With IO bandwidth on the order of 100KB/s, the Input Log for a one-hour execution totals only 360MB, which comfortably fits in a dedicated DRAM buffer.

5.4. Using CADRE

Finally, we examine how to use CADRE. Typically, we use CADRE to replay after a hardware fault is found. In this case, the Memory Log and a register checkpoint are used to restore a checkpointed state. Then, re-execution proceeds by replaying the Input and CPU Logs. Test access ports (e.g., JTAG) are used to read data out of each component. To examine the microarchitectural state in more detail, it is possible to transfer the checkpointed state and logs to a logic simulator. Using the simulator, an engineer can view any waveform in the system. This dump-to-simulator technique is identical to that used in existing systems.

As indicated in Section 5.3, CADRE can be made to support almost unbounded replay periods. This is done by copying the contents of the main memory to a reserved block of "checkpoint" DRAM or to disk, and executing without generating any Memory Log. CADRE can then potentially execute for hour-long periods. This enables schemes for detecting long-latency faults, such as running periodic self-consistency checks in software.

The original motivation for CADRE was to develop a mechanism to assist in system bring-up and test. However, given CADRE's modest overheads, we propose to incorporate it in deployed systems as well. Deploying CADRE in production systems provides hardware vendors with a powerful tool to debug customer-site failures. The customer could send the CADRE checkpoint preceding the crash to the vendor, who could then use it to reproduce the fault exactly using in-house hardware and simulators. The checkpoint includes memory state and Memory Log, the checkpointed register state, and the Input and CPU Logs. The idea is similar to the current use of software crash feedback agents that help software developers identify bugs in deployed software.

6. Related Work

Removing Sources of Nondeterminism. Mohanram and Touba [15] look at the problem of deterministic transmission over source-synchronous buses for the case when the bus clock rate is faster than the core clock rate of the receiver. They propose a slow trigger signal that is synchronized with both clocks. The transmitter and receiver ensure determinism by synchronizing the times of message transmission and reception with the trigger signal.

For globally-asynchronous locally-synchronous clock domains, the Synchro-tokens [9] scheme was recently proposed. In this scheme there are handshake and data signals that are synchronized with each other. After a clock domain receives a handshake signal, it starts its clock, processes the data, and then pauses its own clock. Buses connecting clock domains achieve less than half the throughput, and incur a latency that is at least four times that of synchronous buses. Lastly, this scheme is prone to deadlocks.

Using Logic Analyzers to Log Signals. The most related work is the work on the Golan platform for the verification of Pentium-M [12] (explained in Section 5.3). It uses logic analyzers like the Agilent-8045 [19] to log all the inputs at the processor's pins. The logic analyzer saves the value of the signal along with the clock cycle at which it was latched into the domain of the core clock or bus clock. A typical logic analyzer has 64 channels and logs around 64MB worth of data. It has sophisticated interfaces to view and analyze the data. The Golan platform also has a mechanism to transfer the state to an RTL simulator for further debugging.

Tsai *et al.* [20] propose a scheme to provide deterministic replay for real-time programs. They start out with two processors running in lockstep. Upon a triggering condition or periodically, the second processor freezes itself and hence, it contains the checkpointed state. After that, a logic analyzer starts logging all the signals, interrupts, traps, and exceptions on the processor memory bus, which can be used to replay the execution from the checkpointed state.

7. Conclusions

This paper presented a cost-effective architectural solution to the problem of cycle-accurate deterministic execution on a board-level computer. We characterized the sources of nondeterminism in current computers and showed how to address them. In particular, we presented a novel technique to circumvent one of the most difficult sources of nondeterminism, namely the timing of messages on source-synchronous buses crossing clock-domain boundaries. The proposed solution completely hides this nondeterminism at the cost of a small latency penalty and modest hardware.

Extending a four-way multiprocessor server with the resulting CADRE architecture enables cycle-accurate deterministic execution of one-second intervals with modest buffering requirements (around 200MB) and minimal performance loss (around 1%). The intervals with deterministic execution can be extended to minutes and logging overhead will remain reasonable for many workloads. Such long intervals are a substantial improvement over the tens of milliseconds between checkpoints in current schemes. Overall, CADRE significantly enhances hardware debugging. Moreover, its cost-effectiveness may enable its use in systems in the field.

References

[1] Intel VTune performance analyzer. http://www.intel.com/vtune.

[2] Intel 82801EB I/O Controller Hub 5 (ICH5) and Intel 82801ER I/O Controller Hub 5 R (ICH5R) Datasheet, 2003.

[3] Intel E7525 Memory Controller Hub (MCH) Datasheet, 2003.

[4] F. Bacchini et al. Verification: What works and what doesn't. In *DAC*, page 274, 2004.

[5] L. Bening and H. Foster. *Principles of Verifiable RTL Design*. Kluwer Academic Publishers, 2 edition, 2001.

[6] B. Bentley. Validating the Intel Pentium 4 microprocessor. In *Dependable Systems and Networks*, pages 493–500, 2001.

[7] P. Dahlgren, P. Dickinson, and I. Parulkar. Latch divergency in microprocessor failure analysis. In *ITC*, pages 755–763, 2003.

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

[9] M. W. Heath, W. P. Burleson, and I. G. Harris. Synchro-tokens: Eliminating nondeterminism to enable chip-level test of globally-asynchronous locally-synchronous SoC's. In *DATE*, 2004.

[10] http://public.itrs.net. International technology roadmap for semiconductors, 2002.

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

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

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

[14] N. Lynch. *Distributed Algorithms*. Morgan-Kauffman Publishers, 1996.

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

[16] 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.

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

[18] 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.

[19] Agilent Systems. Agilent technologies E8045B analysis probe system for the Intel Pentium 4 processor in the 775-land package, 2004.

[20] J. J. P. Tsai, K. Fang, H. Chen, and Y. Bi. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Trans. Software Eng.*, 16(8):897–916, 1990.

[21] D. Vahia. Personal communication. *Sun Microsystems*, June 2005.

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