The Embedded Reliable Processing System (TERPS) — A Robust Architecture that Achieves Forward Progress in Near-Continuous Electromagnetic Interference

Cagdas Dirik, Amol Gole, Samuel Rodriguez, Hongxia Wang, and Bruce Jacob

Electrical & Computer Engineering Dept
University of Maryland — College Park
www.ece.umd.edu/~blj • blj@umd.edu

Technical Report UMD-SCA-2004-11-01 — November 2004

We present a fault-tolerant computer architecture that significantly reduces the threat of electromagnetic interference (EMI) by employing an aggressive hardware checkpoint and rollback recovery mechanism that is transparent to application software. Our mechanism periodically checkpoints processor state by saving it into a special safe-storage device; EMI detection results in the mechanism recovering the processor state from a previously checkpointed state and resuming execution at that point. The presence of EMI results only in loss of performance dictated by the EMI duration, and our mechanism ensures forward progress as long as EMI events are separated by a minimum time (e.g., at least 5.12 μs for our prototype processor). We present several different configurations for the handling of fault-tolerant I/ O that provide designers freedom to balance fault-tolerance and complexity considerations and, at the minimum, require only an additional OS interrupt handler that performs device reconfiguration after rollback recovery.

We present a proof-of-concept prototype fabricated through MOSIS and validated by direct injection of EMI through the processor's clock pin—a disturbance that would cause existing systems either catastrophic failure or, at the very best, an inability to make forward progress. We demonstrate, in an actual physical system, the ability to continue functioning correctly even in the face of debilitating EMI, albeit at a much slower pace.

The performance overhead of our mechanism is reasonable (5–6% overhead when checkpointing every 128 processor cycles), and even our simplest I/O configuration (the one with the fewest assumptions and that requires no changes to I/O hardware and software other than the addition of an operating system interrupt handler) guarantees that all I/O data older than 3.96 μs is safe. We also measure maximum I/O downtime after occurrence of EMI: for a system with a single UART I/O device, we show that this downtime is only 7.96 μs.

1 Introduction

EMI affects digital circuits in various ways. A powerful enough burst of electromagnetic interference (EMI) can cause misinterpretation of data, clock, and even power and ground references that can result in chip-wide failure. Continuing developments in chip fabrication technology result in increasingly sophisticated integrated circuits, but as feature sizes of integrated circuits decrease, the circuits' susceptibility to electromagnetic interference (EMI) increases. Consequently, integrated circuits will, in the future, face substantial problems from either electromagnetic disturbances or intentionally-generated EMI from malicious sources.

Moreover, beyond the chip in question, every component in a system becomes a potential entry point for EMI, and once noise is in the system, traditional protection schemes can be rendered ineffective. For instance, if one uses rad-hardening to protect a CPU against direct exposure to EMI, some other component in the system, such as an exposed wire or non-rad-hardened chip, can act as an antenna and ultimately inject noise into the rad-hardened chip through its pins, thereby subverting the chip's protection mechanism. Applications include any complex systems that expect harsh noise environments.

A wide range of techniques can be used to protect the storage and transmission of data, but few techniques are available to protect the processor from chip-wide failure caused by EMI disturbance. In typical computer systems, the processor is one of the components most susceptible to EMI-induced errors because it usually runs at much faster speeds compared to other system components. The tight timing margins involved in processor operation make the processor prone not only to EMI-induced voltage fluctuations resulting in spurious logic values, but also to more subtle EMI-induced timing violations.

We observe that it is far easier to detect correctly the mere *presence* of unwanted electrical noise than it is to determine correctly that any unwanted noise actually *disrupts processing*—i.e., determining whether or not a system is operating correctly (and, consequently, whether or not a rollback should occur) is extremely difficult, and it would be far easier to decide to roll back a system's state upon the mere *possibility* of incorrect operation. The reason that systems do not perform aggressive rollback in this manner is simply because the cost of doing so is high: writing system state to a hard drive can take seconds; writing system state to a solid-state disk, while faster, can still take tens of milliseconds. If, however, the cost of checkpointing can be driven down to a few microseconds or less, then one could in fact afford to checkpoint almost continuously and roll back aggressively whenever EMI is suspected. In other words, if the cost of rolling back system state is low enough, then one can afford to roll back even for false positives.

That is the premise behind *TERPS, The Embedded Reliable Processing System*. By checkpointing in hardware directly from the processor, we only require the register-file contents and a small amount of memory and I/O state. State is transferred to a "safe storage" device (for instance micro vacuum tubes) over a high bandwidth, low latency interface such as through-silicon vias or any other fast interface. We assume that the presence of EMI corrupts the whole processor, except for EMI detectors and a small subset of control logic necessary to control checkpoint/rollback-recovery. Backups of the processor state critical to its operation are checkpointed and stored inside an external safe storage device that is more tolerant of EMI. By ensuring that the checkpoints define a precise state of the processor, EMI recovery simply involves reloading the processor state with a valid previously checkpointed state and continuing execution from that point. Although the results of a handful of instructions that were corrupted by EMI are inevitably discarded by the processor, recovering from a checkpointed state ensures that the processor resumes from a very recent point in the past and retains all the results of the execution up to that point. This saves the processor from losing a significant portion of its data, or experiencing significant down-time, as would be the case if it is forced to shutdown or reboot, or even load a stored checkpoint from a hard disk, because of EMI.

This is a system-level approach to fault-tolerance that allows us to focus EMI protection strategies on a much smaller subset of the whole system: i.e., using a checkpoint/rollback-recovery mechanism pushes the requirement of EMI tolerance from the shoulders of the whole system onto those of the safe storage and associated control logic; consequently, this significantly reduces the susceptibility of the processor to EMI-induced transient faults. The scheme requires only nominal additions to an existing processor core to support our fault tolerant hardware checkpoint/rollback mechanism, leaving the designer free to apply our approach to any existing microarchitecture. Additionally, our hardware mechanisms are fully transparent to the application software and transparent to most of the operating system, making it easily applicable to different architectures.

We present several different mechanisms to handle fault-tolerant I/O, the one area that requires a modicum of operating system support. Supporting I/O is a non-trivial problem because the checkpoint/rollback mechanism that TERPS uses relies on the ability to reexecute instructions; while this presents no problem for memory semantics (rewriting the same value to a location does not alter system behavior), I/O often have operational side-effects (e.g., move an armature one step further, open a valve one step wider, etc.) that are not intended to be duplicated should an instruction need to be reexecuted. We present a range of I/O implementation options that give the designer the choice to modify the operating system, the microarchitecture, or the system's I/O device and peripherals to achieve better fault-tolerant I/O handling at the expense of additional redesign time and system complexity. We show that, even if the designer is not willing to redesign the I/O controller and/or peripheral and uses only the base TERPS configuration, any I/O data older than a specified threshold time can still be guaranteed as safe. Even for our simplest configuration that makes the fewest assumptions and requires no change to the I/O hardware and software other than the addition of an operating system interrupt handler, we can guarantee that I/O data older than 3.96µs are guaranteed to be processed and that our processor only suffers a very short I/O downtime of 7.96µs every time EMI occurs. Even our simplest configuration is a much better option compared to enduring a complete system shutdown, or reboot, or system-state reload from disk, after every EMI occurrence.

The performance overhead of our mechanism is reasonable (5–6% overhead when checkpointing every 128 processor cycles), and even our simplest I/O configuration guarantees that all I/O data older than 3.96µs is safe. We also measure maximum I/O downtime after occurrence of EMI: for a system with a single UART I/O device, we show that this downtime is only 7.96µs.

Lastly, we present a physical proof-of-concept by fabricating a prototype processor chip and prototype safe-storage chip through MOSIS, validating the robust system through direct injection of EMI via the processor's clock pin. This level of disturbance would cause existing systems, even rad-hardened chips, either catastrophic failure or, in the very best scenario, would render the system unable to progress. We demonstrate, in an actual physical system, the ability to continue functioning correctly even in the face of overwhelming EMI, albeit at a much slower pace. We show that our prototype processor assures forward progress even in the presence of EMI as long as EMI events are separated by at least 5.12µs, which is a good representation of typical EMI events [34]. The availability of our processor sharply increases with increasing separation of EMI events and asymptotically reaches 100% availability. Moreover, if one uses faster processors and higher bandwidth backup interfaces, the system scales—the overheads improve, and the system can tolerate even more frequent disturbances.

2 Background

2.1 Electromagnetic Interference (EMI)

As transistors decrease in size due to continuing developments in process technology, the inherent capacitances in these transistors also decrease. With the decrease in capacitance, the amount of electrical charge involved in transistor switching is also decreased. Consequently, the energy required to disturb the switching process is reduced, making it possible to perturb a circuit by using increasingly lower EMI power levels. This situation is exacerbated by the normal practice of reducing the supply voltages and increasing the chip frequencies of new designs, resulting in even tighter noise and timing margins, making the circuit more susceptible to EMI.

EMI can be coupled to electronic systems through cables, printed circuit board traces, bond wire interconnects and even internal chip metal lines for power, ground, clock and data signals -- all of which behave as receiving antennas [7]. EMI signals, when superimposed on existing signals on the chip, can be interpreted as spurious state changes on logic devices [21]. Studies have shown that low power level RF coupled into a system's clock network can upset internal state [32]. Additionally, EMI can change the timing behavior of the signals in a chip, which may also result in timing violations that cause internal latches and flip-flops to work incorrectly.

Previously, typical sources of EMI or radio frequency interference (RFI) were unintentional sources of electromagnetic pollution like overhead high voltage lines, lightning events, radar devices, powerful radio transmitters, wireless network devices and other such sources. But recently, significant concern has been focused on intentionally generated EMI from malicious sources. Bethune addresses the subject of criminal EMI and EM terrorism, defined as "the intentional malicious generation of electromagnetic energy to induce noise or high-level disturbances into electrical or electronic systems with the intention to disrupt, confuse, or damage these systems for criminal or terrorist reasons" [4]. Bethune also reports the capability of off-the-shelf equipment available from RadioShack to generate EMI fields strong enough to affect medical equipment inside ambulances. This introduces serious risks for military equipment, safety- related automotive systems and medical equipment embedded systems that are not specifically designed to be EMI tolerant.

The power levels and frequency range for which circuits are more susceptible to intentional EMI have been studied recently [6] [3] [7]. Specifically, Fiori investigates the effects of RF interference on the input ports of a 0.7μm CMOS device with frequencies in the 20MHz–1GHz range and power levels up to 15dBm; they observe dynamic failures in the form of variations in input pad propagation delay and static failures when pad output signal levels strayed from the high or low range. They show that even less powerful RFI can cause propagation delays and crosstalk-induced delays on wires and can deteriorate signal integrity to the point of failure [7].

Lastly, although electronic equipment can be protected to a certain degree by shielding, filtering and other techniques, these measures are often expensive due to post production costs and are considered infeasible for volume applications [6]. Designing robust ICs will go a long way in helping to solve these problems.

2.2 Related Work

The reliability of computing systems has always been an important consideration, and a significant body of research exists on the subject. Randell provides an insight to different reliability issues including types of faults, fault-tolerance techniques, and examples of fault tolerant systems [18]. He classifies hardware component failures by duration (permanent or transient), extent (localized or distributed), and value (fixed or varying erroneous results).

Redundancy is one of the most widely-used fault-tolerance techniques, and different types of redundancy like space, time, information and algorithmic redundancy are possible [16]. Franklin investigates ways to implement redundancy techniques for superscalar processors and uses under-utilized resources in the system [8]. REESE detects transient faults using time redundancy and adds a small number of extra functional units to keep the overhead low [15]. A special form of space and time redundancy is utilized in the DIVA architecture [33] [1]. This elegant fault-tolerant solution includes a simple checker processor that works alongside the core processor and doublechecks its result. This approach solves a whole range of faults while also reducing the burden of design verification.

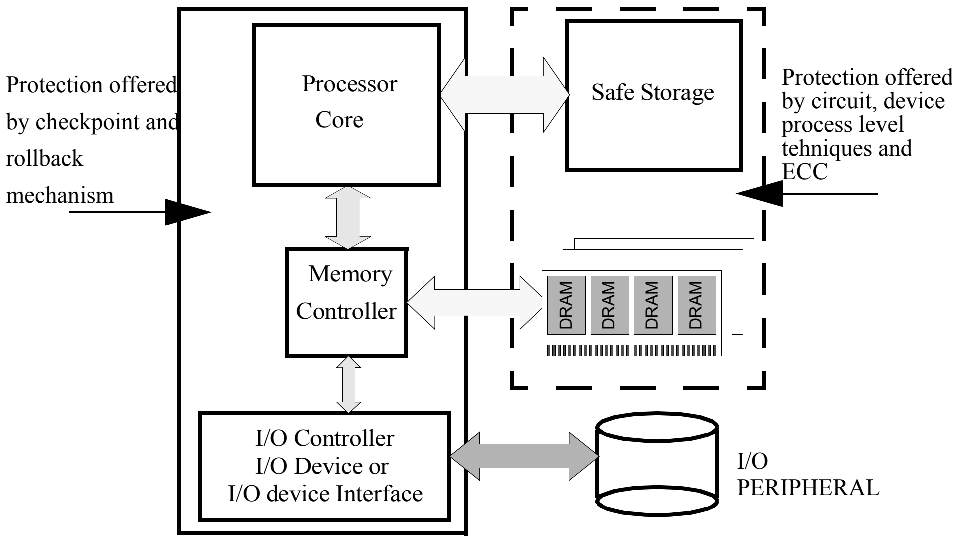

Protection offered by checkpoint and rollback mechanism
Processor Core
Safe Storage
Protection offered by circuit, device process level tehniques and ECC
Memory Controller
DRAM
DRAM
DRAM
DRAM
I/O Controller
I/O Device or
I/O device Interface
I/O PERIPHERAL

Figure 1: TERPS system block diagram.

In general, fault tolerance can be achieved using some form of redundancy with error-recovery algorithms. Once an error is detected, error recovery (in the form of forward or backward recovery) can be used to correct the mistake [18]. An example of forward error recovery mechanism is error-correcting code (ECC) where redundant information embedded into the data allows detection and correction of errors without retransmission. Checkpoint/rollback-recovery is a form of backward recovery technique that has long been a commonly applied fault tolerance approach. Depending on the time and data-integrity requirements, checkpoint/rollback-recovery is implemented either in software or in hardware. The JPL-STAR computer is one of the most significant and earliest systems that adopted checkpoint/rollback recovery, where it is implemented in software [2]. Koo and Toueg present a distributed software algorithm that uses checkpoint/rollback recovery in distributed systems [13]. In this research we target embedded systems that often operate under real-time constraints that prohibit significant delays in recovery. These constraints motivated us to use a hardware-assisted backward error recovery scheme and instruction retry for TERPS.

Instruction retry is used for rapid recovery from transient faults and is seen in many systems including the IBM 4341 processor [5], C.fast [31], the IBM ES/9000 Model 900 [25], the UCLA Mirror Processor [26], and a processor with micro-rollback [28][27]. Shared memory multiprocessors implementing checkpoint/rollback are also presented by Wu [35], Prvulovic [17], and Sorin [24]. TERPS differs from these approaches in that our approach still works in the event of virtually chip-wide failure due to EMI disturbance.

Finally, checkpoint/rollback was proposed by Hwu and Patt for branch misprediction and exception handling in out-of-order processors, where they propose cost-effective algorithms for performing checkpoint repair with very little time overhead [11]. Smith and Pleszkun also present a mechanism wherein instructions can be retried and reexecuted, in their novel structures for implementing precise exceptions in pipelined processors [22]. These works provide the foundation for performing checkpoint/rollback at the granularity of processor instructions.

The various fault tolerant architectures that have been previously proposed have mainly concentrated on protecting systems from single error transient faults, and not from focused intentional EMI that induces chip-wide failure (e.g., our previous work [32] shows that when EMI couples to a CPU's clock network, it can potentially corrupt every stored bit on the chip). TERPS differs from these systems by operating under the harsher assumption that EMI-induced faults corrupt virtually the whole processor (although DIVA could potentially do this also, it

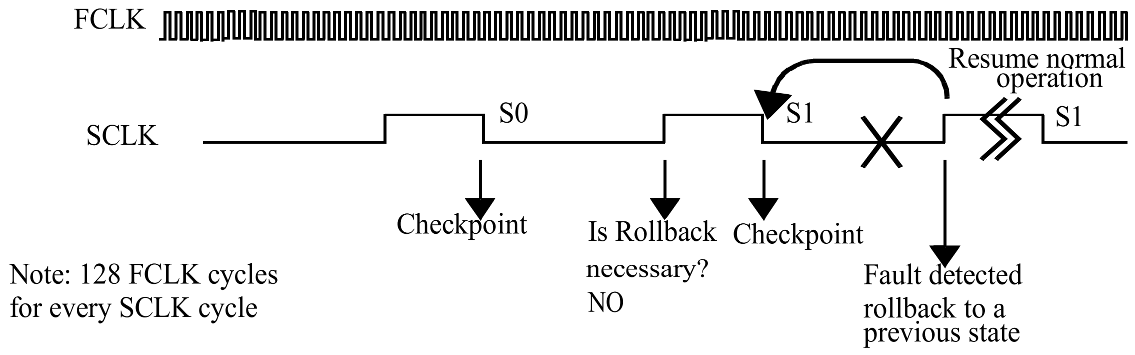

FCLK
SCLK
S0
S1
Resume normal operation
S1
Checkpoint
Is Rollback necessary? NO
Checkpoint
Fault detected rollback to a previous state
Note: 128 FCLK cycles for every SCLK cycle

Figure 2: Simplified checkpoint - rollback mechanism.

requires making the checker processor part of DIVA to be EMI tolerant. This subset of the processor is much bigger compared to the small fraction of processor control logic that we require to be EMI tolerant). TERPS achieves fault-tolerance by borrowing the checkpoint/ rollback concept and applying it to a software-transparent hardware scheme.

2.3 *Scope of Study*

A wide range of techniques is available to protect the storage and transmission of data, but few techniques are available that completely protect its processing. We therefore focus our study on protecting the processor's operation from EMI under the assumption that EMI corrupts the processor core except for a small subset of the control logic controlling the checkpoint and rollback events. We assume that the DRAM system is protected accordingly through mechanisms such as ECC, and we explain how to justify the assumption of a safe-storage device and the associated control logic in the processor. We also focus our study on systems-on-chip (SOC) processors targeted for embedded systems, where the designer has control over the design of the whole system; for this, we present a range of I/O implementations to provide fault tolerant I/O.

3 TERPS architecture

EMI affects digital circuits in various ways. A powerful enough EMI burst can cause misinterpretation of data, clock, and even power and ground references that can result in chip-wide failure. The fault tolerant approaches we have presented in the related work section have aimed at solving a limited number of faults within the system and hence are not directly applicable as a solution to this problem. TERPS is a system-level fault-tolerance approach that addresses the issues related with widespread EMI-induced faults with little performance overhead. It allows recovery from such faults without having to reboot and is fully transparent to application software and nearly all activity of the operating system. In this section, we describe the TERPS architecture and the details of the checkpoint/rollback mechanism used to achieve fault-tolerant operation. We also provide an intuitive discussion of the correctness of the design.

3.1 *Checkpoint/Rollback-Recovery*

Figure 1 shows the TERPS system block diagram. It represents any typical computer system except for the presence of the special safe-storage memory device connected to the processor through a dedicated, high-bandwidth bus. As mentioned, our proposed system architecture protects the CPU and memory-controller operation from EMI by using a hardware checkpoint/ rollback mechanism. We also propose different mechanisms to include the I/O in this sphere of

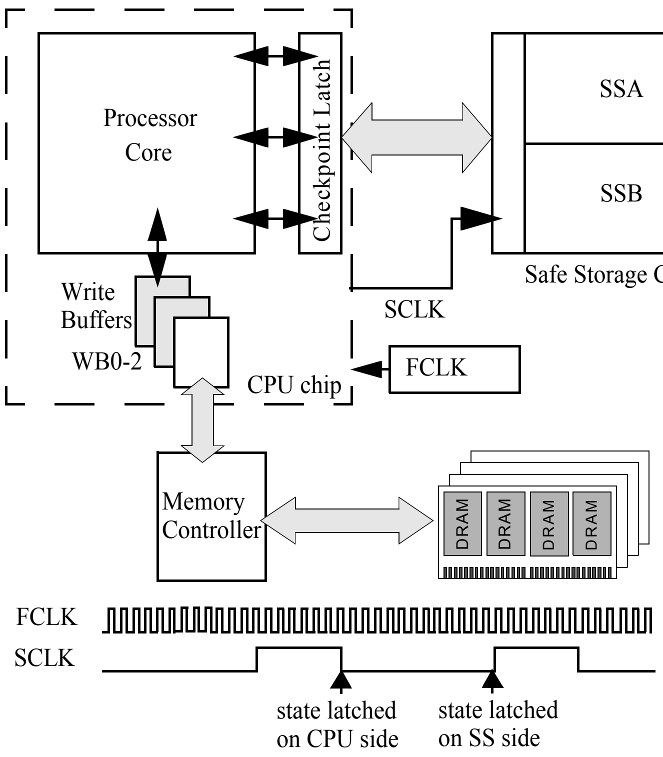


Figure 3: TERPS Architecture.

protection. The rest of this section shows the details of the checkpoint/rollback mechanism; the problem of I/O handling is discussed in detail in a later section.

A simplified version of the checkpoint/rollback operation is shown in Figure 2. In the figure, FCLK is the fast internal clock used by the processor, and it is used to generate a much slower clock (SCLK) that triggers checkpoint and rollback events. For the rest of the discussions, we assume that 128 processor cycles happen during one SCLK cycle (a decision we justify in later sections). A snapshot of the processor state is recorded at every checkpoint, which happens at every SCLK falling edge. An EMI check is done at every SCLK rising edge to determine if it is necessary to initiate a rollback. If EMI has been detected, a rollback is initiated, wherein the processor pipeline is flushed and its state is reloaded from the safe storage. In the figure, the presence of EMI (shown as an X) causes a rollback that reloads state S1. After rollback, the processor resumes operation starting from the reloaded state. By doing checkpoint/rollback, the disruption caused by the occurrence of EMI is minimized, and only a handful of recent results are discarded by the processor; this compares favorably to normal situations in which the CPU, at best, performs a checkpoint at the much longer time granularity allowed by solid-state hard drives or, worse, spinning disks. The TERPS mechanism ensures forward progress of the processor as long as the EMI disturbance is not continuous (i.e. for our prototype processor, forward progress is assured if EMI events are separated by at least 5.12μs).

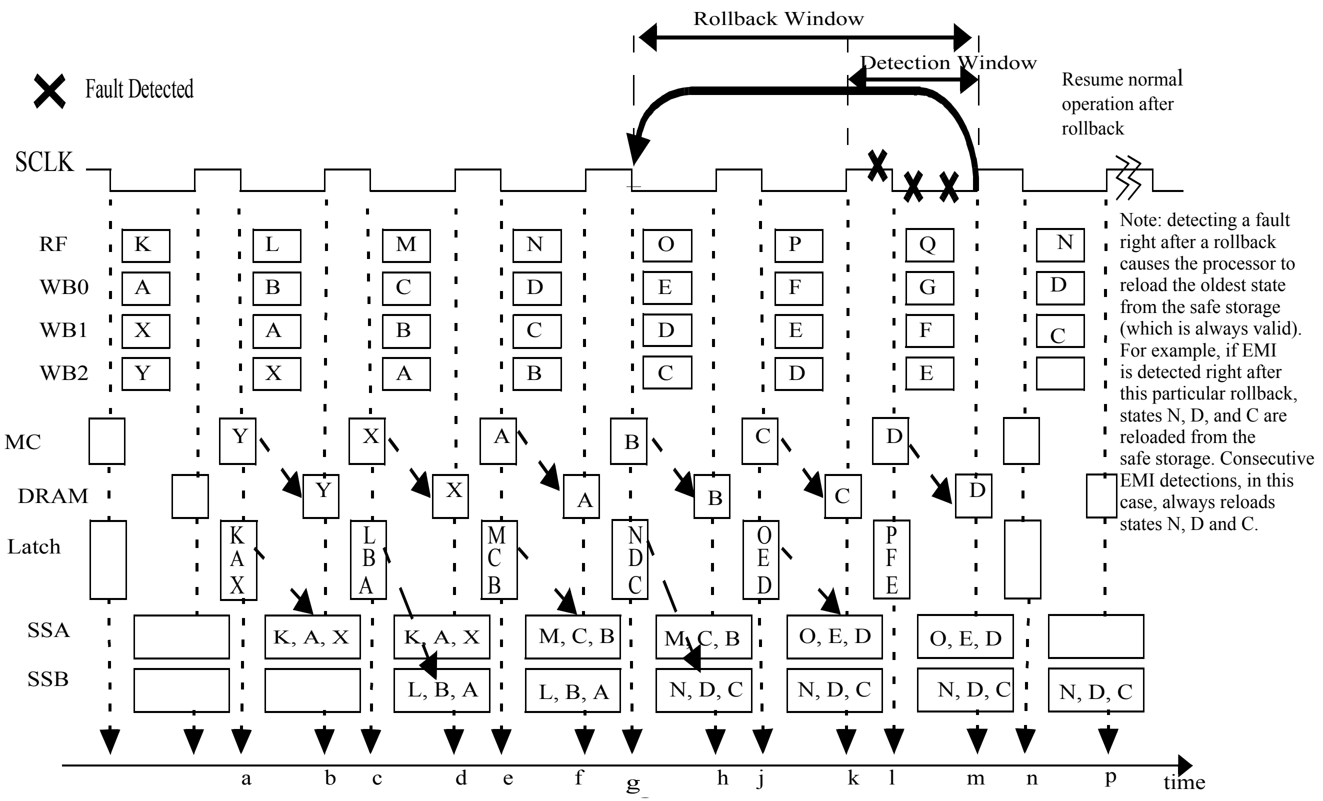
Note that, so far, the description has used multiple simplifying assumptions—specifically that checkpoints can be recorded instantly (on the SCLK falling edge), and that EMI occurrences are detected without delay. These assumptions are unrealistic, and they serve only to facilitate explanation of the basic concepts of TERPS operation. Figure 3 introduces more realistic detail. The figure shows the TERPS architecture (without I/O), including all the mechanisms that allow practical implementation of checkpoint/rollback. These mechanisms are the processor checkpoint latch, the multi-level safe-storage, and the write buffers. These will be discussed one by one.

The first simplifying assumption was the ability to instantly (i.e. in one processor cycle) transfer the checkpointed processor state to the safe storage. This is not feasible, because, as we show later, the safe storage is designed using various techniques that trade off speed for EMI tolerance. Consequently, the safe storage will not be able to keep up with the tighter timing requirements of the processor. We solve this problem by inserting a CPU checkpoint latch in between the processor and the safe storage to ensure that the data is valid for a significant fraction of an SCLK cycle—long enough to satisfy the safe storage's timing requirements (the processor transfers its state to the checkpoint latches at a point that coincides with the falling edge of SCLK; the safe storage then latches this data at the SCLK rising edge). The same argument is applicable for the reverse direction wherein the safe storage provides the data during rollback recovery.

With the addition of the CPU checkpoint latch, checkpoint/rollback-recovery becomes more complicated when we remove the second simplifying assumption that an EMI event is detected instantly. With the checkpoint latch in place, the system has to make a decision during the SCLK rising edge whether to save the new checkpoint data into the safe storage (in the case of no EMI detected), or to perform rollback by retrieving the old checkpoint data stored in the safe store (if EMI has been detected). A delay in EMI detection (i.e. realizing that EMI has happened multiple cycles after it has corrupted data—a window of time potentially large enough to corrupt a checkpoint) may result in the safe storage being overwritten by data corrupted by this undetected EMI. In such a scenario, by the time the EMI is properly detected, the safe storage no longer contains a valid checkpoint to return to, making it impossible for rollback to work. In reality, even perfect EMI detection may still result in latching corrupt data if the EMI occurs during a time that violates the hold-time window of the safe storage.

These problems are solved by implementing two banks of safe storage that store the state from the two most recent checkpoints, thereby ensuring that at least one valid checkpoint always exists in the safe store. With the more realistic assumption that the EMI detection delay is at most one SCLK cycle, a corrupt write caused by EMI overwriting one bank of the safe storage is not disastrous, because rollback will always recover the older state. This assumption provides enough time for the system to correctly determine whether EMI has occurred and ensures that no two consecutive corrupt writes to the safe storage will occur. Consequently, an older checkpoint state will always be valid, and rollback can be performed correctly. Another way of looking at it is that the process of recording new checkpoints is always done speculatively, and the system can recover by keeping an older checkpoint value intact in case the speculation was wrong. With the EMI detection delay assumed to be at most one SCLK cycle, the speculative write of a new checkpoint also means that the previous checkpoint has graduated and is not speculative anymore because it is assured that no EMI was present to corrupt it. This way, the safe-storage always contain both a non-speculative checkpoint that the system uses to recover from misspeculations, as well as a speculative checkpoint.

Lastly, we describe how TERPS deals with instruction retry. All the instructions executed by the processor are performed under the assumption that no EMI is going to happen; if this assumption is proven wrong, the system must take steps to undo the unwanted results and recover back to a precise state [22]. Though this state will vary from architecture to architecture, in TERPS it consists of the program counter (PC) of the instruction that is to be restarted, the contents of the register file (RF), including control registers, and the state of the memory system before this instruction. TERPS easily recovers register file changes by checkpointing a copy of all registers necessary to define a precise state. However, the problem of keeping memory precise is more difficult to implement because of the impracticality of storing copies of the entire memory system into the safe storage. To solve this, we implement a multiphase commit protocol by maintaining a window of memory store instructions in a series of memory write buffers to delay the store data before they



Rollback Window
Detection Window
Fault Detected
Resume normal operation after rollback
SCLK
RF
WB0
WB1
WB2
MC
DRAM
Latch
SSA
SSB
K, A, X
M, C, B
O, E, D
L, B, A
N, D, C
a b c d e f g h j k l m n p
time
Note: detecting a fault right after a rollback causes the processor to reload the oldest state from the safe storage (which is always valid). For example, if EMI is detected right after this particular rollback, states N, D, and C are reloaded from the safe storage. Consecutive EMI detections, in this case, always reloads states N, D and C.

Figure 4: Checkpoint Rollback Recovery. The TERPS checkpoint rollback recovery mechanism can be explained using SCLK as a reference. The results of instructions executed during every period of SCLK are represented as K, L, M, N, O, P and Q. The contents of the RF, write buffers, checkpoint latch, memory controller and safe storage are also shown, along with DRAM updates. At every checkpoint, i.e. falling edge of SCLK, the contents of RF, WB0, and WB1 are written to the latch, the content of WB2 is sent to the memory controller, and buffered stores are promoted to higher-level write buffers. Over time, the safe storage and DRAM are updated with this data. At rollback , the processor state will be reloaded from the oldest checkpointed state stored in the safe storage. An example in this figure shows the system detecting a fault in the interval [k, m] and rolling back to its state at checkpoint g. The oldest checkpointed state is stored in SSB at this time and therefore, the contents of PC, RF, WB0 and WB1 are reloaded from SSB. Right after rollback, the state of the processor is exactly the same as when point g was first seen.

permanently modify memory. These store instructions are not released to the memory system until it is assured that they will not be flushed from the system due to an EMI-induced rollback. The contents of the write buffers in addition to the DRAM state is sufficient to present to the processor a precise view of the memory system.

Store instructions currently being executed are stored in a level-0 write buffer (WB0). Successive checkpoints record a copy of these buffers in the safe storage and, at the same time, promote buffered memory stores to higher-level write buffers. Since we store two different checkpoints in the safe storage at any one time, our system uses three levels of write buffers to implement correct multi-phase commit (one for the stores presently being executed and one each for stores that were executed during the two previous checkpointed state). Store instructions residing in the highest level write-buffer (WB2) are permanently committed to memory at the start of the next checkpoint cycle. This process is shown graphically in Figure 4 and is described in detail in the next subsection.

3.2 Correctness of Design

The principles of checkpoint/rollback are similar to those of handling branch mispredictions or exceptions in out-of-order pipelines, where some in-flight instructions have to be flushed, and execution is restarted from another point. In both cases, the system needs to completely mask out any results from the flushed instructions. In the case of checkpoint/rollback-recovery, detection of a fault causes in-flight instructions to be flushed and execution to be restarted from the reloaded checkpoint state—a point that the processor has passed before. TERPS provides a way to recover correct and precise system state in a manner that is fully transparent to the instruction stream. For any checkpoint/rollback-recovery mechanism to function properly, it is necessary and sufficient to satisfy the following conditions:

- *Precise Checkpointing.* All instructions preceding a checkpoint in the instruction stream have completed, and their results must be reflected in the processor state specified for that checkpoint. At the same time, all instructions occurring after the checkpoint must not have any effect on the state specified for that checkpoint.
- *Precise Rollback.* On rollback, effects of all instructions preceding the instruction boundary defined by the reloaded checkpoint are reflected in the register file and memory while the results of all instructions logically following this instruction boundary are excluded as if these instructions have never been fetched.

These are described in more detail below.

Precise Checkpointing

For any architecture implementing checkpoint/rollback-recovery, the checkpoints should be precise, so that rolling back to any one of these checkpoints results in a state that adheres to the processor sequential architectural model. To implement precise checkpointing, the following two conditions should be satisfied: (i) the state changed by all instructions preceding the instruction indicated by the checkpointed PC must be reflected in that checkpoint, and (ii) the state changed by all instructions following and including the instruction indicated by the checkpointed PC are not reflected in that checkpoint.

Fulfilling these conditions for an in-order pipelined processor is quite straightforward. On checkpoint, the PC of the next-to-complete instruction in the pipeline, the RF and the pending store instructions should be checkpointed. Checkpointing the PC of the next-to-complete instruction ensures that the instructions preceding it would have already completed and updated the RF or write buffers while instructions after it have not changed the state. On a checkpoint, Figure 4 shows our mechanism writing a copy of the PC of the next-to-complete instruction, RF, WB0 and WB1 to the checkpoint latch and in addition, sending the contents of WB2 to the memory controller (MC). The contents of the checkpoint latch and the MC are written to the safe storage and the DRAM, respectively, sometime before the start of the next checkpoint. Lastly, any instruction occurring after a checkpoint will not introduce any changes to the snapshot stored by this checkpoint, ensuring that precise checkpointing is performed correctly. This is easily extended for out-of-order pipelines because these pipelines employ mechanisms that ensure precise execution, like reorder buffers (ROB) [22] or register-update-units (RUU) [23]. These mechanism can provide the checkpoint hardware with sufficient information to define precise checkpoints.

Precise Rollback

In TERPS, a fault detected within a detection window always causes a rollback to the same checkpointed state, as shown in Figure 4. This creates a rollback window that defines the minimum lifetime of an instruction executed by the CPU. No instruction is committed to the system state (released from the system) before it is out of this rollback window. In the TERPS architecture, we declare an instruction to be released when its result is out of the safe storage or, in the case of a store instruction, written to the DRAM. For example, consider in Figure 4 instructions fetched and executed between times **a** and **c**. The results of these instructions are represented as state **B** for store instructions and state **L** for ALU instructions.

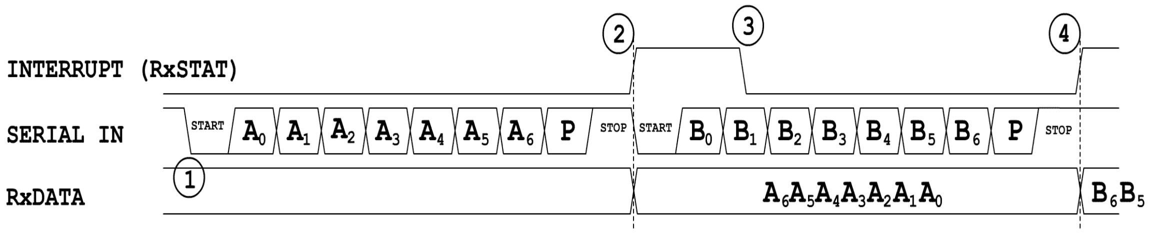


Figure 5: UART receive timing diagram.

(i) Regarding ALU instructions, updates to RF by these instructions are written to safe storage **B** (SSB) at time **d**. At time **h**, SSB will be overwritten by new values represented as **D**, and instructions fetched and executed in the interval [**a**-**c**] are then considered to be released permanently. The time interval between points **c** and **h** is exactly a rollback window.

(ii) If any of these instructions were store instructions, the results are in WB1 at time **c**, in WB2 at time **e**, in MC at time **g** and will be written to DRAM safely by time **h**, where the store instruction is considered as released from the system.

Delaying the release of instructions by a rollback window ensures that the first execution of any instruction is considered speculative until it is verified that no faults occurred in the system during the execution of this instruction. If a fault is detected during execution, instructions within the present rollback window are flushed, and the system rolls back to a previous instruction defined by the oldest state checkpointed in the safe store. Therefore on rollback, instructions that logically occur after the instruction boundary defined by the reloaded checkpoint appear to the system to be unexecuted and unfetched as well. Also on rollback, the contents of the RF and write buffers are reloaded from the safe storage to present the processor with a precise view of the register file and system memory. In Figure 4, for example, consider checkpoint **g** in the program execution. At this point the contents of RF are stored as **N**, and the contents of WB0, WB1 and WB2 are **D**, **C** and **B**, respectively. The processor continues to execute, so that during the interval [**g**-**h**], WB2 updates the DRAM system with **B** and, between **j** and **k**, the DRAM is updated with contents **C**. When the system detects a fault during interval [**l**-**m**], it rolls back to checkpoint **g** in the instruction stream. During rollback, the RF is reloaded with **N** from SSB and the write buffers WB0 and WB1 are reloaded with contents **D** and **C** respectively. Also, note that MC state **B** has been committed and will not be reloaded. In the case of EMI corrupting a commit to memory (like the DRAM write at interval [**j**-**n**] of states **C** and **D**), the state of the memory as seen by an instruction after a rollback will still be correct because correct copies of the data are reloaded from the safe storage into the write buffers. The eventual commit of these reloaded write buffers into DRAM overwrites the corrupt data with correct versions.

Therefore on rollback, the effects of all instructions preceding the checkpoint **g** in the instruction stream are properly reflected in the register file and memory. Lastly, the speculative execution of instructions fetched during the interval [**g**-**f**] do not cause a problem because they are not allowed to update the system state. This ensures that TERPS implements a precise rollback.

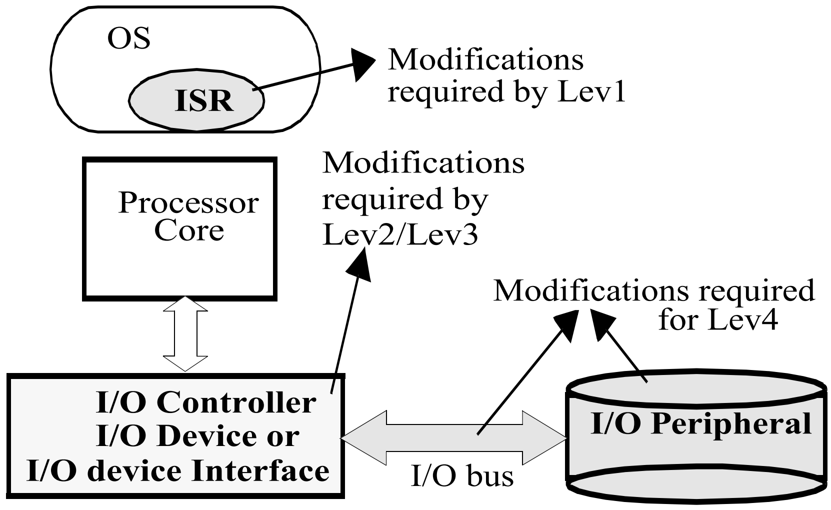

OS
ISR
Modifications required by Lev1
Processor Core
Modifications required by Lev2/Lev3
Modifications required for Lev4
I/O Controller I/O Device or I/O device Interface
I/O bus
I/O Peripheral

Figure 6: TERPS components modified by our fault tolerant I/O implementations.

4 Fault Tolerant I/O

4.1 Differences with Memory

The previous section discussed some details of the TERPS architecture and how it is able to recover from EMI properly by retrieving backups of known-good state. However, the previous discussions have only considered the interaction of the CPU with the safe-storage device and the memory system without giving any specific considerations regarding the handling of I/O devices in the system. This section addresses how the TERPS architecture handles I/O and its inherent differences with memory.

Some I/O devices behave the exact same way as normal memory such that reads and writes to a location can be reexecuted without any problem. Consequently, these types of devices can use the mechanisms that have been previously discussed. Most I/O devices however, have side-effects that will not be exhibited by a memory device. For example, an I/O device receiving new input data will modify its current state without processor intervention. Because of this, two reads from the same I/O address with no intervening stores may result in two different values. Also, writes to the same I/ O address may possibly conflict if the I/O device is allowed to independently modify its contents. A write to an address followed by a read may also result in a different value being read because of how the I/O is allowed to modify its own state.

We ground the discussion in reality by giving an example I/O device: A 16450 universal asynchronous receiver-transmitter (UART) serial controller toggles a status bit and interrupts the processor when new data is received. A single read to the I/O register containing the new data has a side-effect wherein the status bit is automatically deasserted (to reflect the fact that it has been read by the processor) [29]. A UART is a useful device to consider because it can represent both the case where data losses can be tolerated and the case where even minimal data loss results in failure. For simplicity of explanation, we use a UART similar to the commercially available 16450 chip [29]. This type of UART handles data one word at a time and requires processor intervention in between data points. More advanced UARTs like the 16550 [30] implement 16-byte FIFO buffers as a holding tank for data to minimize processor interaction. The concepts discussed here readily extend to these and other, more sophisticated, types of I/O devices.

The UART interrupts the processor whenever new data is received, prompting the processor to execute an interrupt service routine to handle the receive operation. This receive operation is shown in Figure 5. Point 1 shows the detection of a start bit signifying the start of a receive cycle; Point 2 shows the UART has finished assembling the whole data word and interrupts the processor; Point 3 shows the interrupt being deasserted eventually (either automatically as a side effect, or manually by the device driver); and Point 4 shows the start of another receive cycle.

4.2 I/O Scope of Study

Interrupt-driven I/O is more difficult to implement correctly, and so we treat that mechanism here; the device interrupts the CPU whenever it requires handling. The concepts presented here are applicable to polled I/O schemes as well, which is easier to guarantee due to the operating system's explicit management. TERPS can also be extended to implement DMA I/O by treating the DMA controller as a second processor and applying the checkpoint/rollback mechanism to both processors at once. Also, TERPS currently implements memory-mapped I/O, and therefore all I/O devices reside in the processor's memory space. Although this is the case, the fault tolerance techniques we present do not exclude the possibility of using I/O mapped I/O and can be extended to support it. The main benefit of memory-mapped I/O is that no separate write buffers are necessary for I/O writes since these are treated in the same way as memory writes. Lastly, for the sake of brevity, the schemes we discuss only show examples for I/O receive operations since the concepts discussed are directly applicable to I/O transmit operations.

4.3 Fault Tolerant I/O Implementations

Different I/O applications have varying requirements for fault-tolerance. Those with less robust requirements might use TERPS simply to avoid having to undergo a long reboot process whenever the system is corrupted by EMI. These applications might be able to tolerate discarding a few data samples and incurring a slight loss in the quality of the result. On the other hand, applications with more robust requirements might require TERPS to handle I/O without losing any data. Inevitably, applications with stringent requirements for data integrity require more mechanisms to achieve an acceptable degree of fault tolerance—either in hardware, in software or in both.

To accommodate this wide spectrum of I/O requirements, we present a range of implementation options for I/O fault-tolerance. Our approach gives the system designer the freedom to adapt to the fault tolerance requirement of a specific application. The more design modifications a designer is willing to implement, the more robust the resulting system becomes. Applications with simple requirements can make use of the basic TERPS configuration, and increasing requirements need incremental additions to the hardware and/or the operating system. These choices give the designer the freedom to make tradeoffs between system complexity and the desired fault tolerance that is best suited to a particular system.

For the rest of the discussions, we first define the baseline TERPS configuration as the one illustrated in figure 1, with an unmodified standard UART representing the I/O controller connected to an unmodified external I/O peripheral. This represents the simplest implementation of I/O and relies only on the basic checkpoint/rollback-recovery mechanism provided by TERPS.

Figure 6 shows the parts of our system that are explicitly involved in I/O handling. We also show which parts of the system potentially require modifications for our different implementations. Our level 1 (Lev1) implementation uses only the baseline TERPS configuration plus the addition of an interrupt service routine (ISR) in the operating system. Even with this simple implementation, we can guarantee that any I/O datum that has been

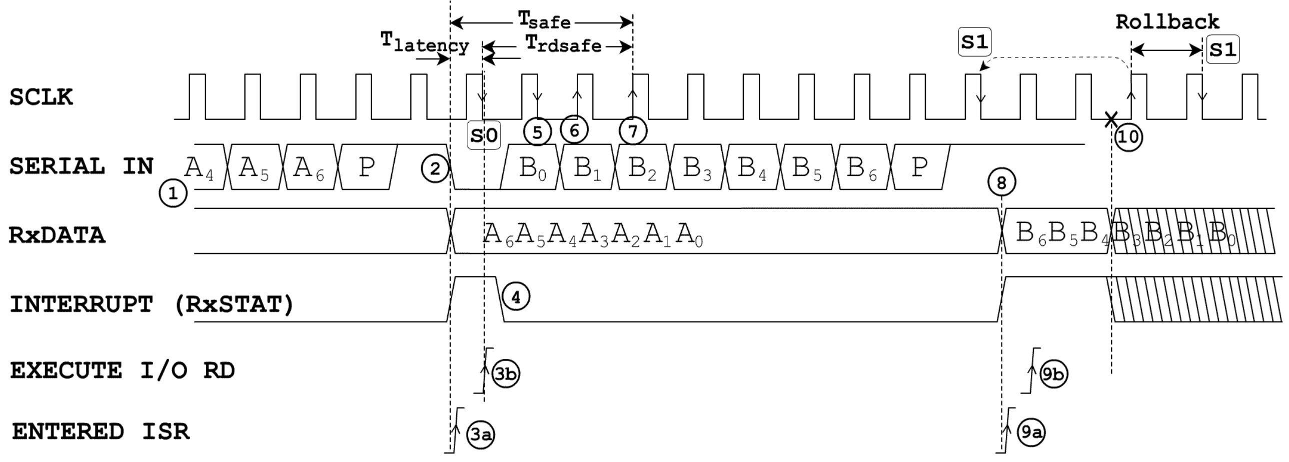


Figure 7: UART receive operation for Level 1 I/O fault tolerance.

received by the I/O controller is safe (from the point of view of the CPU) after a very short time interval has passed (e.g. in the order of microseconds). The downtimes resulting from EMI occurrences are also very small even for this implementation (e.g., a fraction of a millisecond). Our level 2 (Lev2) implementation requires changes in the I/O controller that make it aware of our checkpoint/rollback mechanism. This enables the inclusion of the I/O device state as part of the checkpoint state, resulting in better guarantees and shorter downtimes. Our level 3 (Lev3) implementation requires changes in both the I/O controller and the operating system I/O device drivers. It requires the I/O controller to be redesigned to minimize access side-effects to facilitate instruction retry. In addition, it also requires redesign to make it more EMI tolerant to ensure that it retains its state even in the presence of EMI. These redesigns allow the processor to reexecute I/O instructions and consequently, any data that has been received by the I/O controller is guaranteed to be safe and eventually processed by the CPU. The first three implementations are CPU-centric and focus on protecting data that the processor is aware of when the data has been received by the I/O controller. In contrast, our fourth level (Lev4) is able to protect everything including the actual I/O peripheral by virtue of using transmission protocols to protect the passage of the data from the peripheral to the I/O controller. We next discuss these implementations in much more detail.

Level 1: No Additional Hardware Modifications

The first implementation assumes nothing other than the basic TERPS configuration and is suitable for designs where the designers are unwilling to modify existing I/O devices and peripherals. This configuration offers some guarantee of fault-tolerance by providing a time guarantee wherein I/O data older than a certain threshold time, T_{safe}, are guaranteed to be handled properly, and only data younger than T_{safe} are discarded if EMI corrupts the system. Although it provides the least guarantee out of all the possible approaches, this configuration still has much better fault-tolerance compared to a typical non-fault-tolerant system and is the least costly and complicated because it can utilize existing implementations for the I/O controller and the actual I/O peripheral.

Additionally, since no special fault-tolerant precaution was used in the design of the I/O controllers and peripherals, TERPS assumes that the occurrence of EMI totally corrupts the I/O device, requiring device reinitialization. This is performed by an OS interrupt handler responsible for reinitializing the I/O devices that is invoked after every system rollback. As a consequence, this reinitialization widens the time window wherein the I/O device is not responding and results in a greater probability of losing data. Although designers choosing this option still experience downtime and possible data losses during EMI attacks, the downtime length is still vastly reduced, compared to a rollback using SSD or HDD, let alone a full system reboot.

Figure 7 shows a timing diagram demonstrating this approach, using the UART as an example. In the figure, a new datum is assembled by the UART at point 2, prompting the UART to interrupt the processor. The processor enters its ISR at point 3a and executes the critical I/O instruction, in this case the I/O data read, at point 3b, storing the datum in the CPU register file. This datum is only guaranteed to be safe starting at point 6, a point where any future rollback will always come back to a state that has executed the previous I/O read (which in this case is denoted by state S0). An EMI occurring any time before this point will force the system to reload a state that does not recognize the newly received datum. This is demonstrated by an occurrence of EMI at point 10, rolling back to the saved state S1 that has not performed the I/O read and therefore does not contain the desired datum, in this case, B[6:0].

Referring to figure 7, to determine the worst-case time guarantee, it is important to clarify that a received datum is guaranteed safe whenever the I/O read instruction retrieving the datum has been executed without any possibility of the results being discarded by a rollback. This is the case when the critical I/O instruction has passed through two full checkpoint operations, guaranteeing that any future rollback restores a processor state that has already finished executing the I/O instruction, thereby guaranteeing the safety of the read data. We call the maximum time interval where the critical I/O read instruction waits for two full checkpoint intervals to pass T_{rdsafe} (Note that T_{rdsafe} is exactly the same duration as a rollback window). For the TERPS, a checkpoint cycle starts at the falling edge of the SCLK and finishes at the next rising edge. The maximum wait time will occur when the I/O instruction just missed being included in the start of a checkpoint (i.e. it occurs right after a falling edge), and it therefore has to wait a full SCLK cycle before it is included in a checkpoint. The maximum T_{rdsafe} is shown in the figure—it starts right after an SCLK falling edge and finishes at the third SCLK rising edge (the first rising edge is the end of the missed checkpoint, and the next two rising edges are for the two actual checkpoints needed to save the state). This interval takes 356 processor cycles for the TERPS and is 3.56μs long for a 100MHz processor.

In addition, there will always exist a delay between the first appearance of the data and the time when the critical I/O read instruction is executed. We call this delay $T_{latency}$, and it includes both the actual interrupt latency of the processor and any preambles executed by the interrupt service routine. Although a maximum value for the interrupt latency can be determined, $T_{latency}$ is variable because of the dependence on the preambles executed by the ISR. In our case, we assume the use of 40 processor cycles total for $T_{latency}$, enough to account for the interrupt latency and most preambles that need to be executed. $T_{latency}$ therefore is 0.4μs long for a 100MHz processor.

T_{safe}, being the sum of $T_{latency}$ and T_{rdsafe}, is therefore 3.96μs for a 100MHz processor. Therefore, our time guarantee states that any I/O data present in the system longer than 3.96μs is safe and guaranteed to be processed by the CPU. Only I/O data more recent than 3.96μs will be discarded when EMI occurs, but 3.96μs is a very short period of time in view of typical I/O throughputs.

Lastly, the downtime can be separated into three components: the time when EMI has occurred but is undetected, the time needed for rollback, and the time required to reinitialize the I/O devices. The total downtime is harder to quantify because of the time component dependent on how many I/O devices are being reinitialized and the complexity of each initialization. For our sample system containing a single UART, the maximum total initialization time is four SCLK cycles equivalent to 512 processor cycles (to allow the I/O writes enough time to traverse the write buffers plus all the miscellaneous ISR preambles and postambles).

Assuming the maximum EMI detection latency is 128 processor cycles (as was used in the previous section), and rollback time is 156 processor cycles, the total downtime becomes 796 processor cycles equivalent to 7.96μs for our 100MHz processor. TERPS rollback ensures that processing does not restart from the very beginning but instead at a recent point in the past that retains all the previous results and computations up to that time. This holds true even if the time to reinitialize I/O is high in the case of complex I/O devices.

This situation demonstrates that the basic TERPS configuration still provides some level of fault-tolerance by itself even without modifying the I/O devices. This time guarantee, though, does not guarantee integrity of all I/O data, possibly making it unsuitable for applications with more robust requirements.

Level 2: Checkpoint the Device

The first implementation used I/O devices and peripherals that are unaware of the TERPS checkpoint/rollback-recovery mechanism. If the designer is willing to modify the I/O controller to take advantage of TERPS, the time guarantees given to the designer can be reduced further, especially the device downtime.

Designing the I/O device to respond to the checkpoint/rollback mechanism used by the processor enables inclusion of the I/O device state as part of the system state guarded by the safe storage. By checkpointing device state, the time guarantee, T_{safe}, is freed from its dependence on when the software performs the critical I/O operation. T_{safe} now becomes the worst-case interval between the point when the I/O changed its internal state up to the earliest time when that state can be restored by any future rollbacks. This turns out to be the same as the previous T_{rdsafe}, so we have decreased the total time of T_{safe} by $T_{latency}$. Therefore, T_{safe} now becomes 3.56μs, down from 3.96μs.

More importantly, checkpoints of the I/O device include complete state information that defines the I/O device operation, and restoring this state is enough to reinitialize the I/O. The total downtime is reduced because everything is reinitialized automatically during the rollback process. This prevents the need for the processor to manually reinitialize every I/O device and results in a significant reduction of total downtime. For our sample system with the UART, we reduce the total downtime by four SCLK cycles equivalent to 512 processor cycles, resulting in a total downtime of only 284 cycles, which is only 2.84μs for a 100MHz processor compared to the previous 7.96μs downtime. It is important to note that this reduction is even more significant for systems with more I/O devices, since these will have to take longer to reinitialize if done manually through a device reinitialization handler.

This situation demonstrates that modification of the I/O controller to take advantage of the TERPS checkpoint/ rollback mechanism results in a better time guarantee for data and also a significant reduction in downtime experienced by the system after occurrence of EMI.

Level 3: Circuit Level Modifications to I/O Controller

The main limitation of the first two I/O implementations is the assumption that the I/O device state is totally corrupted by EMI, resulting in discarding data inside these devices and requiring device reinitialization. If we relax this assumption, we can guarantee that any data that has been received by the I/O device is already safe and does not have to wait for time T_{safe}. By using the same techniques that made the safe-storage device more EMI tolerant (as will be discussed in later sections), it becomes possible to assume that the I/O device maintains its internal state even with the presence of reasonable amounts of EMI, although data being currently received by the I/O device can still be corrupted.

The basic requirement of this "safe-data" guarantee is that I/O instructions can be safely reexecuted without any deviation in behavior. In contrast, the guarantees of the previous approaches relied on the critical I/O instructions being safe after time T_{rdsafe}; EMI occurring before the critical I/O instruction is older than T_{rdsafe} forces the processor to rollback its state and overwrite the I/O operation. With the previous assumption that I/O devices are corrupted, the processor cannot correctly reexecute the device handler to retrieve the data. The relaxed assumption of the new approach makes it possible for the processor to reexecute the critical I/O operations, making the time guarantees unnecessary.

To make I/O instructions reexecutable, the I/O device also has to be redesigned to minimize the side-effects involved in its operation. Going back to the the UART receive operation shown in figure 5, typical UARTs automatically deassert their receive interrupt (point 3) when a read from the receive data register RxDATA is performed. Although the processor is only aware of executing a read operation, the end result is a simultaneous read-write. This is a problem of atomicity where side-effects result in committing part of an operation instantly, even though the whole operation has not yet been cleared to commit. This becomes a problem since TERPS is always executing everything speculatively by virtue of the checkpoint/rollback scheme. The side-effect results in lost information that is unrecoverable, preventing the processor from executing the instruction sequence more than once. This problem does not exist in the first two approaches because I/O instructions were not required to be reexecutable.

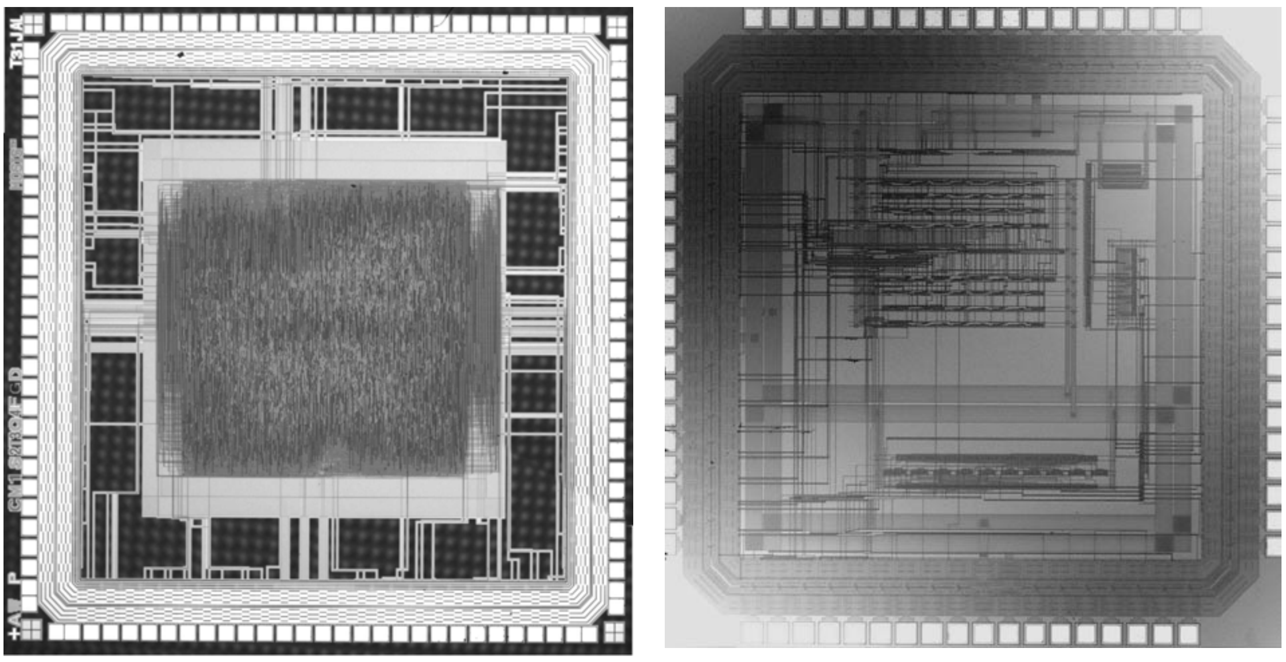
To provide the safe-data guarantee, the designer has to be willing to redesign the device to minimize these side-effects, retaining only those that are not disastrous and can be solved by careful operation and sequencing of the I/O device driver. The device driver becomes responsible for using the TERPS hardware mechanisms properly (especially the memory write buffer) to ensure that, with the previous assumption that the internal device data is safe, TERPS can reexecute the interrupt handler doing the I/O read without any change in behavior, guaranteeing that any data already inside the I/O device is safe and will be processed by the CPU.

Although this guarantee for the received data is given, it does not cover any data currently being received by the UART, and any serial data bits being received will be corrupted by EMI. If an additional assumption is made that the EMI bursts are short and infrequent enough that it allows the UART's majority voting mechanism [14] to offset the detection errors, this scheme guarantees that all the data are received and processed properly without any data losses. The EMI assumptions require that the EMI burst is less than half a bit period and occurs at most once per received word to guarantee the interrupt handler enough time to read the data. These assumptions are reasonable based on typical characteristics of intentional-EMI attacks, both narrowband and wideband [34].

This situation demonstrates that modifying and re-designing the device to minimize its side-effects and to ensure that its internal storage is safe allows the reexecution of I/O operations and guarantees that data already residing inside I/O devices will be handled properly. With the additional assumption of short and infrequent EMI, the system guarantees that all I/O data are processed without any losses. In addition, the downtime is now minimized since the I/O device continues operating directly after the EMI burst. Although the processor itself experiences some downtime because of the rollback process, the I/O devices are already available to respond to further requests.

Level 4: Protocol Changes to Device and Peripheral

For longer bursts of EMI, the previous approach provides no guarantee that future data will be processed properly. If the designer is willing to modify the I/O peripheral, even tighter guarantees can be reached. Extending the previous situation, if the EMI burst is greater than half a bit period and enough to corrupt a received bit, forward error correction methods like ECC can be used to automatically recover from these bit errors. This requires modification to both



(a) Processor Core (b) Safe Storage

Figure 8: Chip die photos fabricated through MOSIS.

the I/O device receiving the ECC-encoded bits, and the I/O peripheral generating and transmitting these bits.

While ECC-encoding might be enough to recover from EMI bursts that span the transmitted word, longer EMI durations will require more extensive protection mechanisms. Using transmission protocols between the UART and the I/O peripheral (i.e. framing, checksums, heartbeats, etc.) will ensure that, even in the presence of very long EMI bursts, the system is able to process I/O data properly by retransmitting lost data.

Usage of data retransmission will go a long way toward guaranteeing the integrity of I/O data, and it provides the highest guarantee of all our implementations. If the designer is willing to modify all the parts of the I/O subsystem, then extremely robust fault-tolerant I/O can be achieved. Although this robust fault-tolerance coverage requires significant modifications, mission-critical systems that have need for this kind of fault-tolerance have a bigger probability of being designed from scratch, making the penalty for modification much less severe. It is also important to note that modification involves only a one-time redesign, after which the design can be reused to the same extent as the old unmodified designs but will now provide much better fault-tolerant guarantees.

4.4 Application Interface

The hardware checkpoint/rollback scheme used by the TERPS architecture is totally transparent to the software (both the operating system and the user application) when handling system memory, in much the same way as out-of-order and/or superscalar execution mechanisms are made software transparent (unless the developer is performing hardware- dependent code optimizations). Software has no idea that our fault-tolerant mechanism is in place performing regular checkpoints and initiating rollbacks when necessary. When accessing I/O, our approach has the operating system assume some of the responsibility in maintaining the fault-tolerance guarantees. Depending on the implementation, the changes to the operating system ranges from the addition of a simple interrupt handler that is invoked after every rollback (to reinitialize I/O

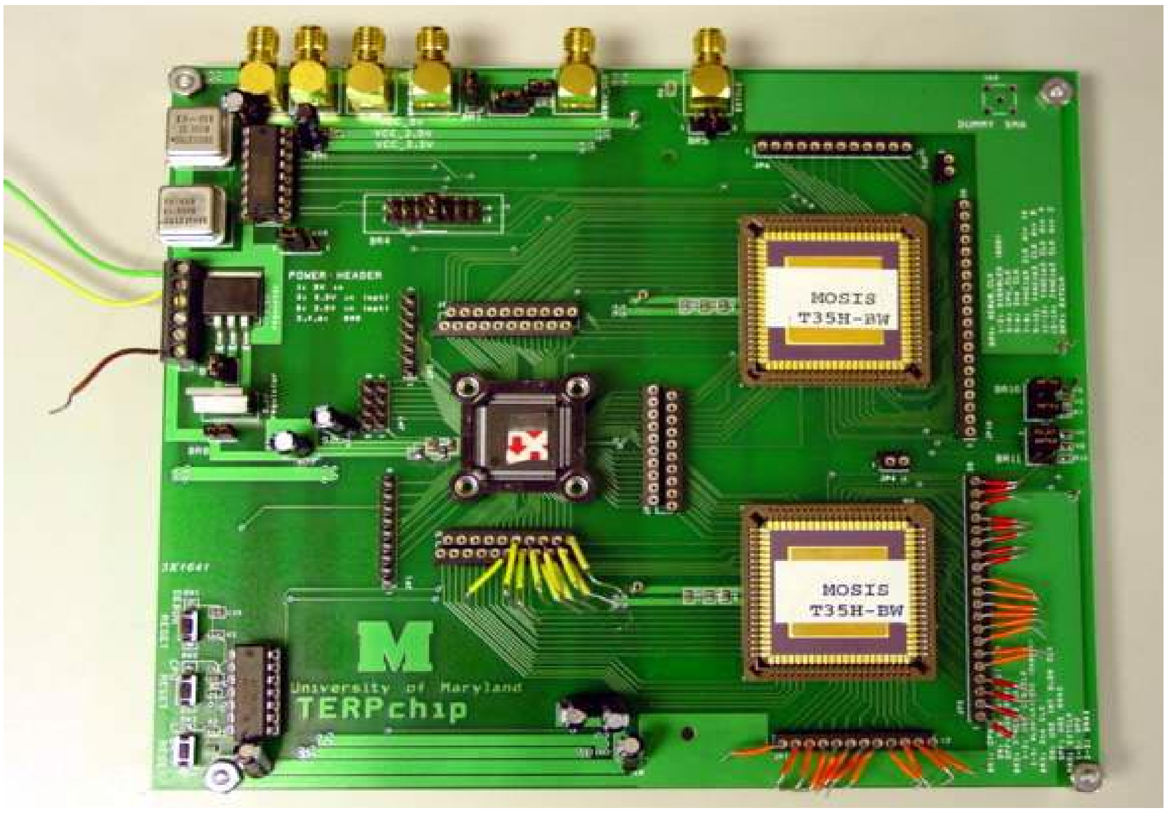


Figure 9: Printed circuit board containing our fabricated processor and safe storage integrated circuits, along with miscellaneous support circuitiy.

devices), to modifications of the I/O device driver making it aware of the TERPS mechanisms for checkpoint/rollback.

5 TERPS Implementation

5.1 CPU

Our initial prototype uses a simple 16-bit RISC with a 5-stage in-order pipeline that is similar to the DLX/MIPS processor [10]. Use of this pipeline was purely for simplicity, and our checkpoint/rollback mechanism can easily be adapted for more complex architectures like superscalar, out-of-order machines.

We fabricated the CPU through MOSIS using the TSMC 0.25mm process, and Figure 8a shows a photomicrograph of the processor chip. We ran our prototype processor at a frequency of 100 MHz, and implemented a step-down circuit that produces 1 SCLK cycle for every 128 FCLK (processor) cycles. These numbers result in a maximum instruction throughput of 128 instructions per checkpoint cycle. We choose to implement 12-entry write buffers for each of the three write buffer levels, to produce a good tradeoff between storage requirements (more entries require a bigger safe storage) and performance overhead (fewer write buffer entries result in higher probability of filling the buffer up, stalling the processor until the next checkpoint is taken). These particular numbers result in a 5–6% checkpointing overhead for our prototype. This performance overhead represents only the overhead occurring during normal operation when no EMI is detected, caused by the stalls due to waiting for write buffer entries and transferring data atomically to the memory controller. The occurrence of EMI results in the loss of 4 SCLK cycles worth of operation, or 512 processor cycles (356 cycles worth of discarded

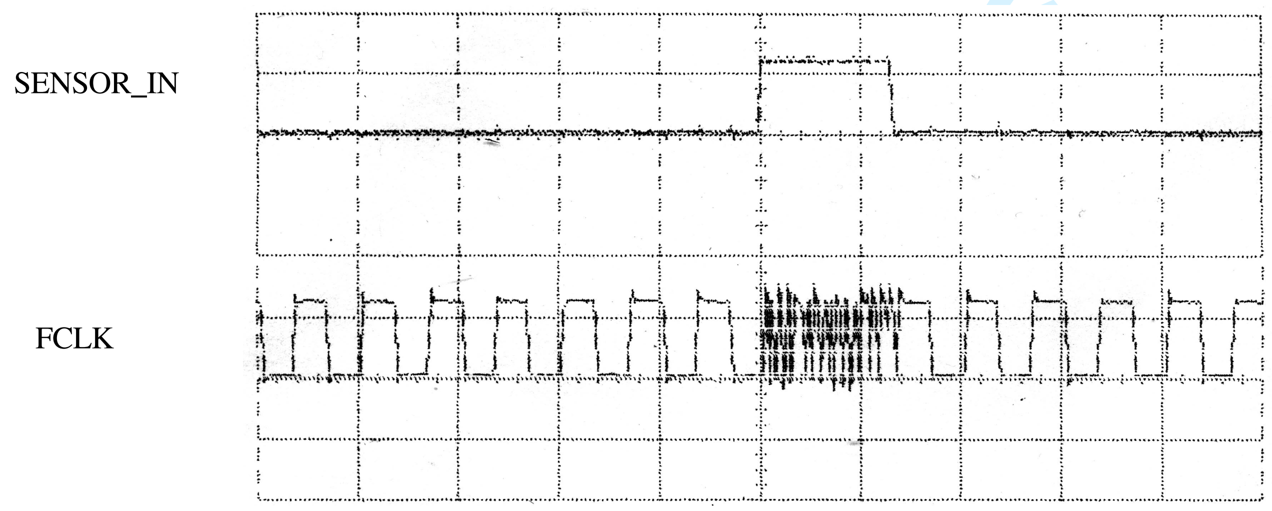


Figure 10: Oscilloscope screenshot showing the system clock (FCLK) signal being perturbed by RFI and the resulting signal from an RF detector.

instructions in one rollback window, and 156 cycles for the actual rollback), which is equivalent to 5.12μs for a 100MHz processor.

5.2 *Safe Storage*

The checkpoint/rollback mechanism works under the assumption that processor-wide EMI-induced faults can be solved by safely reloading its state from the safe-storage. TERPS transfers most of the responsibility of EMI tolerance to the safe storage. This is advantageous because of two reasons: 1) focusing EMI tolerance techniques on a much smaller subset of the design results in a more robust component, and 2) minimal design restrictions are placed on the processor. These reasons make it possible to design a fast, high-performance system that is not slowed down by its fault tolerance mechanisms.

The safe storage is a memory that is specially designed to have significantly better EMI tolerance than the processor by using a variety of architecture, circuit, device, and process-level techniques. Most of these are orthogonal to each other and may be used or left out depending on the level of tolerance required by the system and the willingness of the designer to accept the necessary tradeoffs. Most of these design techniques trade off speed and/or die area for better EMI tolerance. Additionally, the subset of the control logic within the processor associated with controlling checkpoint/ rollback can be made EMI tolerant using similar means. This is still possible because of the very small size of the circuitry involved in controlling checkpoint/ rollback.

We implement our prototype safe-store SRAM using a six-transistor cell to maximize the static noise margin (SNM) of our circuit. The SNM is a good measure of the amount of spurious signal needed at the memory cell inputs to corrupt its state. The SNM of different memory cell configurations has been extensively studied (a good example is [20]), showing that the 6T configuration is the best choice to maximize SNM and is the best choice if higher EMI tolerance is needed. In a production system, one would clearly use memory technology even more EMI resistant, such as micro vacuum tubes or high-capacitance SRAM cells.

The soft-error rate (SER) of SRAMs in the presence of alpha particles has been widely studied [12, 19], and it has been shown that maximizing the stored charge in the memory cell makes it harder for alpha-particles to disturb the state of the cell, resulting in better SER. The most common way to increase this stored charge is to increase the parasitic capacitance of the cell output nodes so that more charge is stored for a given supply voltage. This capacitance can be

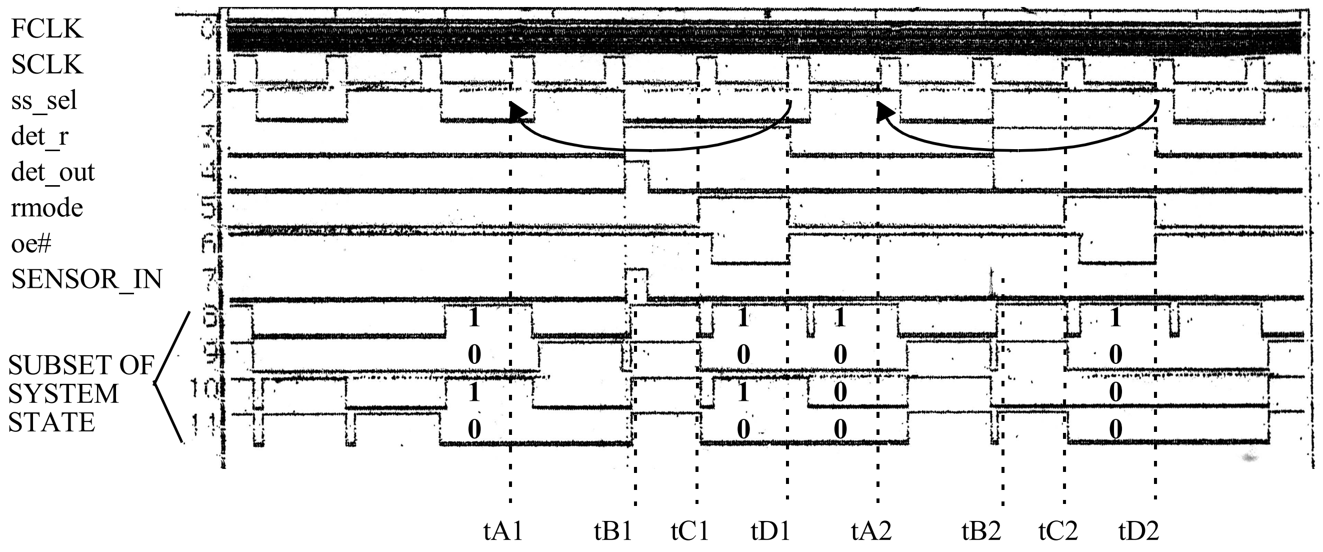


Figure 11: Logic analyzer screenshot of checkpoint/rollback-recovery waveforms showing two occurrences of EMI events and subsequent recovery. The signal SENSOR_IN notifies the checkpoint/rollback control logic of the presence of EMI, initiating necessary steps for recovery. For this sample run represented by this screenshot, at time tB1 RFI is detected and at time tC1 the control logic initiates rollback-recovery where data is retrieved from safe storage and latched at time tD1. The data retrieved at time tD1 is the same data that was backed-up at the safe storage at time tA1, as required by the protocol we explained and demonstrated in Figure 4. Times tA2 to tD2 simply show a second occurrence of RF and the corresponding system recovery. Note that for the sake of brevity we only show a small subset of the system state transferred between the processor and safe storage, and that details regarding implementation-specific control signals (shown in lower-case) are omitted.

increased using device-level techniques that enlarge the cell area to increase the parasitic diffusion capacitances. Hence, higher capacitance is achieved at the expense of a larger cell area. Process-level techniques can also be used, and one example is adding additional layers to create overlap capacitances [9]. In this case, higher capacitance and EMI tolerance is achieved in exchange for process complexity. Achieving better SER by increasing the electrical charge stored in the memory cell also results in better EMI tolerance because it requires higher EMI power levels induce voltages in the system that are large enough to exceed the cell's SNM and corrupt it.

The same principle can be applied to the whole storage system and not just the storage cells. Using transistors with larger areas and powered by higher supply voltages will result in increased charge stored within the system. Consequently, this increased charge will require larger amounts of EMI to corrupt and push around. Since the safe-storage area needs larger transistors and higher supply voltages to increase the stored charge, we fabricate our safe-store using a larger feature size process that is about two generations older than the one used for the CPU. This ensures that the safe-storage EMI tolerance is better than the processor's. Using these techniques, we fabricated the safe storage through MOSIS using the AMI 0.5μm process, and the photomicrograph is shown in Figure 8b.

5.3 *Verification*

We verified the physical functionality and correctness of our complete system by creating a printed circuit board testbed as shown in Figure 9.

We test our system by simulating EMI-induced faults through the direct injection of RF interference into the system clock. As mentioned in section 2.1, we have shown that this kind of interference will cause large, wide-scale, multi-bit errors potentially corrupting virtually the whole processor core [32]. Figure 10 shows a screenshot from an HP 54510A oscilloscope of the system clock when RFI is present and the corresponding signal generated by the RF detection

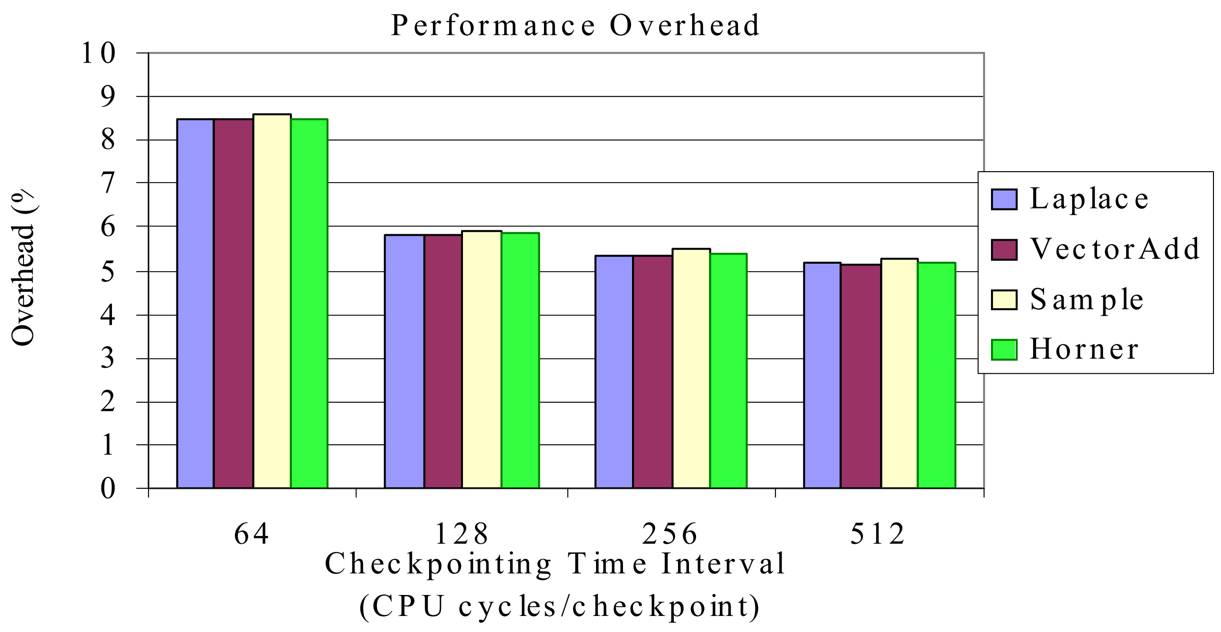


Figure 12: Performance overhead of checkpointing mechanisms .

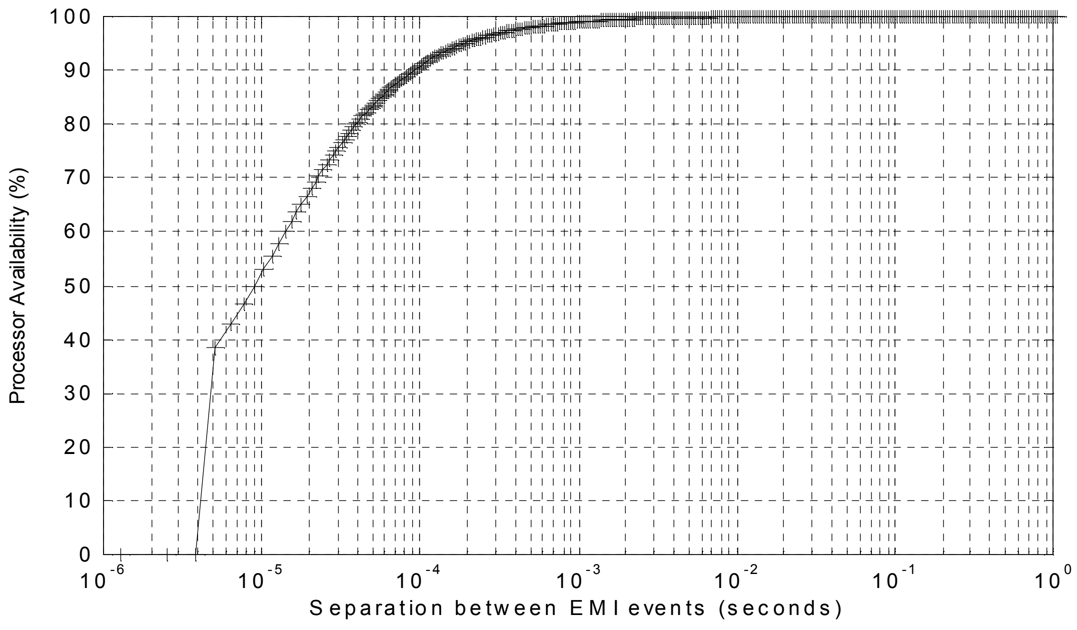


Figure 13: Processor availability vs. separation of EMI events.

circuitry to inform the processor of the presence of EMI. To ease the process of physical testing and verification in satisfying our requirement that a very small subset of the processor involved in controlling our checkpoint/rollback-recovery mechanism is kept safe at all times, we isolate the clock signal being fed to this subset of the circuitry and make sure that it is clean even under the presence of EMI. In a production system, this circuitry could be protected in ways similar to those described in Section 5.2, and by using larger, more robust transistors.

Figure 11 shows a screenshot from an HP 1662AS logic analyzer showing our checkpoint/rollback-recovery waveforms, with the CPU being subjected to frequent EMI disruptions through the clock network—an attack that would cause most systems catastrophic failure. The screenshot demonstrates system behavior that exactly follows the protocol earlier described in Figure 4. The screenshots essentially show the flow of data to and from our safe storage (along with the control signals needed to coordinate these operations) and how they perfectly correlate with our expected data flow, validating the correctness and functionality of our physical prototype. We have also physically verified that, while RF interference corrupts the processor state (observable by tracking the program counter on a cycle-to-cycle granularity and comparing it with the expected value), our checkpoint/rollback-recovery mechanism allows the processor to completely recover from this otherwise fatal event.

5.4 *Sensitivity Analysis*

To give a feel for different design issues, such as different frequencies of checkpointing, this section presents a brief evaluation of our system with respect to several design parameters, performed using execution-driven cycle-accurate models of our design using Cadence Verilog HDL. During normal operation (undisturbed by EMI), our mechanism introduces some overhead because the processor is forced to stall during checkpoints to transfer the write buffer (WB2) contents fully to the memory controller. This overhead is dependent on application behavior, and Figure 12 shows the performance overhead for our processor (measured using different microkernels), for different checkpoint intervals and with write buffer sizes chosen to minimize stalls due to write buffer overflow. For a checkpoint interval of 128 cycles and a write buffer size of 12, our overhead is only 5–6%.

The occurrence of EMI forces the processor to discard recent instructions and spend some time to perform the rollback recovery; consequently, the performance of the processor will depend on how frequently EMI events occur. Figure 13 plots the availability of the processor (which we define as the percentage of time used in performing useful work that results in forward progress) plotted against the average separation between EMI events. For a processor that checkpoints every 128 processor cycles, our processor assures forward progress as long as the average EMI separation is at least 512 processor cycles (e.g. 5.12μs for our prototype processor running at 100MHz. Our mechanism can scale with increasing frequencies as long as the relative speed between the processor and the safe storage is maintained), resulting in an availability of 11%. The availability sharply increases to 57% for an average EMI separation of 12.8μs and asymptotically approaches 100% with increasing EMI separation.

6 Conclusion and Future Work

We have proposed a fault-tolerant architecture that significantly reduces the EMI susceptibility of a system. Our mechanism periodically checkpoints processor state into a special safe storage device and reloads this checkpointed state into the processor after the occurrence of EMI. We describe checkpoint and rollback in detail, including practical mechanisms we have used to solve problems caused by non-ideality of the system. We have also presented different implementations for achieving fault-tolerant I/O, and we provide the designers with a choice on which approach to use depending on the balance between complexity and fault-tolerance required by the application.

For future work, we are currently applying our checkpoint/rollback-recovery mechanism to come up with fault- tolerant versions of commercial microprocessors such as ARM. In addition, we are currently studying the effects and possible advantages of exposing some parts of our mechanism to the operating system. By doing so, it may be possible to offload I/O complexity required to support fault tolerance onto the operating system, resulting in simpler fault-tolerant I/O devices.

The performance overhead of our mechanism is reasonable (5–6% overhead when checkpointing every 128 processor cycles), and even our simplest I/O configuration guarantees that all I/O data older than 3.96μs is safe. We also measure maximum I/O downtime after occurrence of EMI: for a system with a single UART I/O device, we show that this downtime is only 7.96μs.

To prove the feasibility of our fault-tolerant architecture, we have designed and fabricated prototype integrated circuits and printed circuit board testbeds and demonstrated correct functionality of our proposed checkpoint/rollback- recovery mechanism in the face of pulsed RF injected into the clock pin of the CPU.

Lastly, we show that our prototype processor assures forward progress even in the presence of EMI as long as EMI events are separated by at least 5.12μs. The availability of our processor drastically increases with increasing separation of EMI events and asymptotically reaches 100%. This shows that our fault-tolerant architecture still performs useful work even in the presence of frequent EMI disruptions.

7 References

[1] Austin, T. M., "DIVA: A Reliable Substrate for Deep Submicron Microarctecture Design," *MICRO-32. Proc. 32nd Annual International Symposium on Microarchitecture*, 16-18, Nov. 1999, pp. 196-207.

[2] Avizieniz, A. et al., "The STAR (self testing and repairing) computer: an investigation of the theory and practice of fault tolerant computer design," *IEEE Trans. on Comp.* C-20, 11, Nov. 1971, 1312-1321.

[3] Baffreau,S., Bendhia,S., Ramdani, S., Sicard, E., "Characterisation of Microcontroller Susceptibility to Radio Frequency Interference," *Proc. of the Fourth IEEE International Caracas Conference on Devices, Circuits and Systems*, 17-19 April 2002, pp. I031-1 - I031-5.

[4] Bethune, J., Conroy, S.S., "Newsline: The New Cold War: Defending Against Criminal EMI," *Compliance Engineering*, May-June 2001. Available: http://www.cemag.com/archive/01/05/news.html.

[5] Ciacelli,M. L., "Fault Handling on the IBM 4341 Processor," *11th Fault-Tolerant Computing Symposium*, Portland, Maine, June 1981, pp. 9-12.

[6] Fiori, F., "Integrated Circuit Susceptibility to Conducted RF Interference," *Compliance Engineering 17*, no. 8 (2000), pp. 40-49.

[7] Fiori, F., Benelli, S., Gaidano, G., Pozzolo, V., "Investigation on VLSI's Input Ports Susceptibility to Conducted RF Interference," *IEEE International Symposium on Electromagnetic Compatibility*, 18-22 Aug. 1997, pp. 326 -329.

[8] Franklin, M., "Incorporating Fault Tolerance in Superscalar Processors," *Proc. 3rd International Conference on High Performance Computing*, 19-22 Dec. 1996, pp. 301 -306.

[9] Geppert, L., "A Static RAM says goodbye to data errors," *IEEE Spectrum*, February 2004.

[10] Hennessy, J.L. and Patterson, D.A., *Computer Architecture A Quantitative Approach*, Morgan Kaufmann Publishers Inc., 2 nd edition, 1996.

[11] Hwu, W.-M. W., Patt, Y.N., "Checkpoint Repair for Out-of-order ExecutionMachines," *Proc. of the 14th annual International Symposium on Computer Architecture*, June 1987, pp. 18-26.

[12] Ishibashi, K. et al, "An alpha-immune 2-V supply voltage SRAM using a polysilicon PMOS load cell," *IEEE J. Solid-State Circuits vol SC-25*, no.1, pp.55-60, Feb.1990.

[13] Koo, R. and Toueg, S., “Checkpointing and Rollback-Recovery for Distributed Systems,” *IEEE Trans. on Software Engin. SE-13*,(1), Jan. 1987, pp. 23-31.

[14] Motorola, *M68HC11 Datasheet*, http://e-www.motorola.com/files/microcontrollers/doc/ref_manual/M68HC11RM.pdf, 2004.

[15] Nickel, J. B., Somani, A.K., “REESE: A Method of Soft Error Detection in Microprocessors,” *Proc. The International Conference on Dependable Systems and Networks*, 1-4 July 2001, pp. 401 -410.

[16] Peercy, M., Banerjee, P., “Fault Tolerant VLSI Systems,” *Proceedings of the IEEE*, Vol. 81, No. 5, May 1993, pp. 745-758.

[17] Prvulovic, M., Torrellas, J., Zhang., Z., “ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors,” *Proc. of the 29th Annual International Symposium on Computer Architecture*, May 2002, pp.111-122.

[18] Randell, B., Lee, P., Treleaven, P. C., “Reliability Issues in Computing System Design,” *ACM Computing Surveys*, vol.10, issue 2, June 1978, pp. 123 - 165.

[19] Sato, H. et al, "A 500-MHz pipeline burst SRAM with improved SER immunity," *IEEE J.Solid-State Circuits vol.SC-34*, no.11, pp.1571-1579, Nov.1999.

[20] Seevinck, E. et al, “Static-noise margin analysis of MOS SRAM cells,” *IEEE J.Solid-State Circuits*, vol. SC-22, no.5, pp.748-754, Oct. 1987.

[21] Sicard, E., Marot, C., Fourniols, J.Y., Ramdani, M., “Electromagnetic Compatibility for Integrated Circuits,” *Techniques l'ingénieur/Techniques for Engineers*, 2003.

[22] Smith J.E. and Pleszkun, A.R., “Implementing Precise Interrupts in Pipelined Processors,” *IEEE Trans. Computers*, C-37(5), May 1988, pp.562-573.

[23] Sohi, G.S., and Vajapeyam, S. “Instruction issue logic for high-performance, interruptible pipelined processors,” *Proc., 14th Annual Symp. on Computer Architecture*, 1987, pp.27-36.

[24] Sorin, D.J., Martin, M.M.K, Hill, M.D. and Wood, D.A., “SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery,” *Proc. of the 29th Annual International Symposium on Computer Architecture*, May 2002, pp. 123-134.

[25] Spainhower, L. et al., “Design for fault-tolerance in system ES/9000 model 900,” *Proc. 22th Int.Symp. on Fault-Tolerant Computing*, July 1992, pp. 38–47.

[26] Tamir, Y., Liang, M., Lai, T., and Tremblay, M., “The UCLA Mirror Processor: A Building Block for Self-Checking Self-Repairing Computing Nodes,” *Proc. 21st Int'l Symp. Fault-Tolerant Computing*, June 1991, pp. 178-185.

[27] Tamir, Y., Tremblay, M., “High Performance Fault-Tolerant VLSI Systems using Micro Rollback,” *IEEE Transactions on Computers*, Volume: 39 Issue: 4 , April 1990, pp. 548-554.

[28] Tamir, Y., Tremblay, M, and Rennels, D.A., “The Implementation and Application of Micro Rollback in Fault-Tolerant VLSI Systems,” *18th Fault-Tolerant Computing Symposium*, Tokyo, Japan, June 1988.

[29] Texas Instruments, *TL16C450 Asynchronous communications element datasheet*, http://www-s.ti.com/sc/ds/tl16c450.pdf.

[30] Texas Instruments, *TL16C550C, TL16C550CI Asynchronous Communications element with autoflow control Datasheet*, http://www-s.ti.com/sc/ds/tl16c550c.pdf, 2004.

[31] Tsao, M.M. et al. “The Design of C.fast: A Single Chip Fault Tolerant Microprocessor,” *Proc. 12th Int. FTCS*, June 1982, pp. 63-69.

[32] Wang, H.; Dirik, C.; Rodriguez, S.V.; Gole, A.V.; Jacob, B., “Radio frequency effects on the clock networks of digital circuits,” *Electromagnetic Compatibility*, 2004. EMC 2004. 2004 International Symposium on ,þVolume: 1 ,þ9-13 Aug. 2004 Pages:93 - 96 vol.1.

[33] Weaver, C. Austin,T., “A Fault Tolerant Approach to Microprocessor Design,” *Proc. The International Conference on Dependable Systems and Networks*, 1-4 July 2001, pp. 411-420.

[34] Wik, M.W., and Radasky, W.A., “Intentional Electromagnetic Interference (IEMI) — Background and Status of the Standardization Work in the International Electrotechnical Commision (IEC),” *XXVIIth General Assembly of the International Union of Radio Science*, Aug 17-24, 2002.

[35] Wu, K.-L., Fuchs, W. K., and Patel, J. H., “Error recovery in shared memory multiprocessors using private caches,” *IEEE Trans. on Parallel and Distributed Systems*, vol. 1, no. 2, Apr. 1990, pp. 231–24.