Patching Processor Design Errors

Satish Narayanasamy Bruce Carneal Brad Calder

Department of Computer Science and Engineering
University of California, San Diego
{satish, bcarneal, calder}@cs.ucsd.edu

Abstract—
Microprocessors can have design errors that escape the test and validation process. The cost to rectify these errors after shipping the processors can be very expensive as it may require replacing the processors and stalling the shipment.

In this paper, we discuss architecture support to allow patching the design errors in the processors that have already been shipped out. A contribution of this paper is our analysis showing that a majority of errors can be detected by monitoring a subset of signals in the processors. We propose to incorporate a programmable error detector in the processor that monitors these signals to detect and initiate recovery using one of the mechanisms that we discuss. The proposed hardware units can be programmed using patches consisting of the errata signatures which the manufacturer develops and distributes when errors are discovered in the post-design phase.

I. Introduction

Industry spends as much as 50 to 70 percent of its efforts on validation and testing of new hardware [1]. Increasing hardware complexities, and time to market pressures contribute to design errors. Despite significant efforts, some of these design errors find their way into production, after which the cost of dealing with them can be substantial.

If serious errors are found before a chip is shipped, the repair cost includes the engineering time to characterize and repair the errors, the direct fabrication costs associated with a re-spin, including new masks, and, most importantly, the lost revenue attributable to delayed introduction of the product. If serious errors are discovered after shipment, the costs to correct the problem can include all of the above along with the in field replacement costs and a severely tarnished reputation for the company. Recently, 3000 Opteron processors were recalled as they were found to be vulnerable to an error that can produce incorrect results for a particular sequence of floating point operations [2]. Perhaps the most well known design error is the FDIV [3] bug that cost Intel about $475 million as the manufacturers had to provide a replacement to any customer that reported a faulty part. In fact, almost every processor has a good number of design errors in them, which the manufacturers discover after shipment and publish them in the errata sheets [4], [5].

There are two ways to reduce the post shipment costs: (a) prevent errors from occurring in the first place or (b) make it easier/cheaper to correct errors as they are discovered. Our natural inclination, and most of the resources committed to date, are toward prevention. But with so much being spent on prevention already, and errors still getting through, we should also focus on designing mechanisms to patch design errors in the shipped processors. The latter is the goal of this paper.

Currently, it is possible to correct some errors through BIOS patches, by modifying the software, and by reconfiguring the system configuration (eg: motherboard). Such mechanisms often resort to disabling some functionality in the processor to fix an error. For example, a design error in Pentium4 can be overcome by having the firmware (BIOS) disable cache prefetching [6]. Another error in AMD64 processor can be avoided by disabling dynamic power management [5]. Many errors relating to the instructions' execution can be fixed by patching the micro-op generation mechanism [7]. However, executing the patch code every time the error prone instruction is executed can result in significant performance overhead. An alternative to these is to have a well tested checker processor in order to verify the correctness of a complex out-of-order core [8], but design errors in the periphery of a processor core (e.g., coherency) can be difficult to patch with this approach.

In this paper, we propose to include a hardware unit in the processor when the processor is designed, which can be programmed later to patch errors when they are discovered. We do not focus on the initial discovery of the design errors in the processor. Instead, our goal is to patch the errors in the shipped out processors after they are diagnosed by the manufacturer. Our approach can detect the possibility of an error while the processor is executing, and apply the patch only when it is necessary to correct the error.

A key observation makes this possible. By studying the design errors in Pentium4 [4] and AMD64 [5] processors, we find that a majority of errors can be detected by monitoring a well defined set of signals in the processors (e.g., signals used for monitoring the performance, interrupts, exceptions, etc). Hence, the error (discovered by the manufacturer) can be represented using an errata signature, which consists of a set of signals along with their values that are required to detect the error. The error signature can be used to program our proposed hardware units to detect the errors.

We also examine a set of mechanisms to patch an error once it is detected by our programmable error detector. Some errors related to CISC instruction implementation can be patched through instruction stream editing [9]. Errors that are dependent on the occurrence of a specific sequence of events in the processor can be patched by rolling back and re-executing the running threads. We also discuss how hypervisor [10],

[11] support can help patch processor design errors. Through empirical analysis, we find that these mechanisms are capable of patching 78% of AMD64 errors and 69% of Pentium4 errors without leading to degradation in functionality.

II. Current Error Correction Techniques

In this section, we focus on techniques currently used by the manufacturers to patch the errors discovered in the post-design phase.

A. BIOS Patches

The primary purpose of the BIOS is to set up the initial control states in the processor during start up, before the control is handed over to the operating system. Since the processor's control settings are dependent on the configuration of the external devices in the system, usually the BIOS is customized to a particular OEM vendor's system configuration.

Processor manufacturers have found the BIOS (in general, firmware) to be a good place to apply hardware patches, as it lies at a convenient abstraction layer that does not affect the higher level software users and it also has knowledge about the system's configuration. The BIOS can be used to set up the initial control state in such a way that it would circumvent a hardware error. For example, it is possible to overcome a particular hardware error by setting the control bits appropriately to increase the pulse width of the signals (e.g., Erratum #98 in AMD64 Errata [5]).

Perhaps, the most common form of patching using the BIOS are the micro-code patches [7]. In modern processors, certain instructions are already translated to a sequence of micro-operations. Hence, if the hardware error is related to the implementation of an instruction with a particular opcode, then we can fix the bug by just patching the micro-code corresponding to that opcode. A similar mechanism can be used in processors like Crusoe, which have a layer of software called the Code Morphing Software [12] on top of the processor to translate the x86 instructions into the native ISA.

The BIOS has certain limitations. First, we find that a significant proportion of the BIOS patches involve disabling a feature in the processor to overcome the problem. Such fixes can degrade performance or functionality in the processor. Examples for disabling the functionality include disabling the write combining feature in write buffers (Erratum #133 in AMD64 [5]) or disabling a power optimization feature (Erratum #78 in AMD64 [5]).

Second, the BIOS has only limited run-time support and does not have knowledge about all the events in the processor. Though the BIOS micro-code patching will work for patching the error in the implementation of a particular instruction opcode, it is not suitable for patching more complex design errors. Also, the BIOS cannot detect hardware errors during the program execution to take corrective action on demand. As a result, if the error occurs only under special circumstances, executing the patch code for every executed instance of the instruction can result in inefficiencies. For example, Erratum #103 for AMD64 [5] says that the incorrect execution of the `AAM` instruction only occurs under a very specific pipeline condition, so it does not need to be patched every single time the instruction is executed. In Section V, we discuss our hardware patching mechanisms that can detect the occurrence of an error and trigger the execution of the patch only when it is necessary.

B. Software Patches

It is possible to overcome some hardware errors by taking preventive action in the software layer. For example, code generation in a compiler or run-time system can make sure that certain opcodes never appear adjacent to each other in order to avoid a particular sequence of events that trigger an error. Patching a hardware error in software is typically only an option if there is a single operating system and compiler/linker used for the processor. For example, DEC successfully masked hardware errors using link-time optimization for released processors, making sure certain opcode sequences would never occur during execution. This was possible because they were in complete control of the operating system (OSF) and the compiler chain. For mainstream processors, software patching is not as feasible, because of the multitude of different operating systems and compilers supported by them.

C. Re-spin

A re-spin of the chip is both the most costly and the most capable of the post fabrication patching options. Bugs can be fixed during a re-spin, but there is also a chance for introducing new bugs. The costs associated with a re-spin include additional engineering time, the purchase of new masks, and a potentially devastating delay to market. Though re-spinning can completely fix a hardware error, it would involve replacing the customer products, which is a more costly exercise than sending a simple patch to the customer. Further, re-spinning requires significantly more time when compared to just sending a patch to the customer.

III. Taxonomy of Errata

In this section, we identify the common types of processor design errors. We also analyze the importance of the errors that are discovered in the post-design phase.

A. Source of Design Errors

The AMD64 errata sheet [5] reports 63 errors found in the AMD Athlon 64 and AMD Opteron processors (hereafter referred to as AMD64). The Intel Pentium4's errata sheet [4] reports 109 errors. Table I classifies the design errors reported in those errata sheets into various categories. A few of the errors are classified under more than one category. Hence, the numbers do not add up to 100%. For clarity, the table also contains the absolute number of errors (presented inside the braces).

The memory interface category includes the errors in the bus interfaces, cache and virtual memory implementations. About 17.4% of errata in AMD64 and 20.2% in Pentium4 can be classified under this category. This category of errors constitute the highest proportion of errors among all the categories. This shows how the external memory interfacing is error prone.

All the errors that relate to the interaction between multiple processors are classified into the Multi-processor category. Especially, coherence bugs are prominent in this category. This happens to be the next most common form of errors in AMD64, as they constitute about 14.2%. In Pentium4 processors about 5.5% of the errors are classified under this category.

Error Type	AMD64	Pentium4
Memory interface	17.4 (11)	20.2 (22)
Multi-processor	14.2 (9)	5.5 (6)
Power management	11.1 (7)	3.7 (4)
Incorrect error report	11.1 (7)	6.4 (7)
Opcode Implementation	11.1 (7)	16.5 (18)
64-bit extension	9.5 (6)	11.0 (12)
Frequency	4.7 (3)	1.8 (2)
Interrupt	7.9 (5)	1.8 (2)
Exception	6.3 (4)	2.7 (3)
Debug Support	3.1 (2)	11.0 (12)
Hyper-threading (SMT)	NA	11.0 (12)
VT (Vanderpool)	NA	8.3 (9)
Unpatchable	9.5 (6)	2.7 (3)
None of the above	3.1 (2)	0.9 (1)

TABLE I
TAXONOMY OF DESIGN ERRORS IN AMD64 AND INTEL PENTIUM4.

Power management is relatively a new functionality added to the CPU design. We see that about 11.1% and 3.7% of errors in AMD64 and Pentium4 fall under this category. The Pentium4 number is a little lower than the AMD64 number because we classified some of the Pentium4 bugs in this category as Hyper-threading bugs.

Processors incorporate diagnosis functionality to detect certain faults (e.g., faults in memory by using ECC). We found seven errors in both AMD64 and Pentium4 that are due to incorrect error reporting. Incorrect error reporting is a recurring problem in these implementations, but the impact from these errata is usually less than catastrophic as they do not lead to incorrect execution of programs. Errors in this category include off by one counters, mismanaged counter overflows, and extended delays when reporting a condition.

The opcode implementation category contains those bugs that are clearly identified with the incorrect implementation of an instruction with a particular opcode. In x86 ISA implementations it is common for a complex CISC instruction to get incorrectly implemented. For example, the CPUID instruction is incorrectly implemented and can lead to incorrect behavior under some circumstances.

Recent advancement to 64-bit resulted in 9.5% of the errors in AMD64 and 11.0% of the errors in Pentium4. Interesting examples in this category include various incorrect implementations of the CISC based string instructions when they operate on operands that are over 2^{32} in length.

The 'frequency' category contains all those bugs that only occur with specific operating frequencies or clock ratios. A number of bugs in this category occur in conjunction with peculiar but legal motherboard configuration choices. Interrupt bugs are the incorrect implementation of external interrupt handlers, while the exception category represents the implementation bugs that manifest when the chip attempts to deal with internal (CPU originated) exceptions. Incorrect interrupt and exception handling together account for nearly 14% of errors in AMD64, but it is only 4.5% errors in Pentium4.

The next category listed is related to the debugging support provided in hardware. A good number of the debug support bugs had to do with altered execution flow of the application's execution. Various problems involving the single step execution facility are classified under this category. Mismanagement of the data watch-point capability are also common.

We next list the errors seen in Intel's Hyper-Threading(HT) and Vanderpool technologies, which resulted in 12 and 9 respectively out of the Pentium4's total of 109 errors. We did not find any erratum related to the implementation of Pacifica technology in the AMD64 errata sheet [5].

The Unpatchable category consists of those bugs that are not good candidates for architectural patching. For example, errata that are related to voltage levels in the processor not meeting the published specifications are classified under this category.

B. Importance of Design Errors

There are two factors that determine the importance of an erratum: frequency of occurrence and severity of the error. Unfortunately, the errata documents available to us give only a limited view of the frequency of the individual erratum. However, they let us know if it is possible for an error to occur at the customer site or whether it occurs only when the processor is subjected to a contrived in-house testing. Table II, categorizes the percentage of errors into these two categories: Customer and In-house errors.

Clearly, the errors that have the potential to occur at the customer site are important. However, for certain errors, like a very small time slip in the performance counter accounting, the errata sheet mentions that they are not a serious cause for concern. We classify such errors into "Customer; Unimportant" category and the rest into "Customer; Important".

	AMD64	Pentium4
Customer; Important	79.35	74.31
Customer; Unimportant	3.17	0.00
In-house; Plan to fix	11.11	8.26
In-house; No plan to fix	4.76	17.43

TABLE II
IMPORTANCE OF DESIGN ERRORS IN AMD64 AND INTEL PENTIUM4.

For errors found during in-house testing, the manufacturers also indicate whether or not they ever plan to fix the errors. Thus, we further classify in-house errors into two categories based on whether the manufacturer has plans to fix it or not. We believe that errors that the manufacturer plans to fix are also important. By taking the first and the third row in the

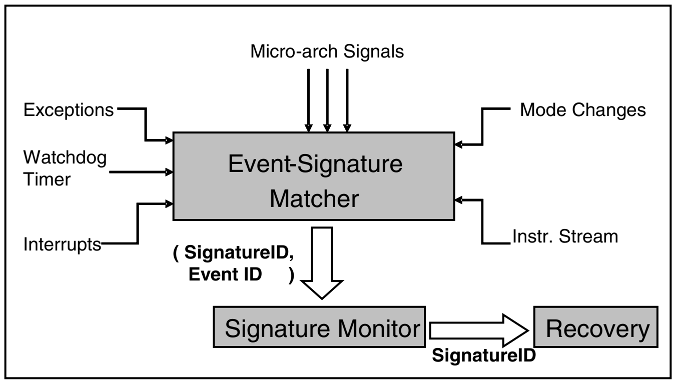

Fig. 1. Patching design errors using errata signatures.

Table II together, we note that approximately 90% of errors are important and a mechanism to patch these errors in shipped processors will be of great value.

C. *Empirical Study on Design Errors*

Similar to our empirical analysis of errata, Aviczienis and He [13] studied the errors in Pentium-II that are discovered in the post-design phase. Their study focused on analyzing the importance of design errors. Their study reports that most of the errors are concentrated in the parts of the processors that are meant for fault tolerance (e.g., ECC). However, we find that in recent processors a significant fraction are due to the new functionalities added to the processor (e.g., Hyper-Threading in Pentium4).

IV. Detection Using Errata Signatures

In this section, we show that it is possible to detect a majority of design errors by monitoring a set of signals that can be determined during design. We propose to include a programmable error detection unit in the processor, which has the capability to monitor a set of signals that are commonly involved in processor design errors. When a new error in the design is discovered after the processor has been shipped out, the processor manufacturer can develop a patch for it. The patch consists of an errata signature.

A. *Patching Hardware Errors using Errata Signatures*

An errata signature is used to program our hardware patching mechanism to detect errors that manifest due to an erratum. An *Errata Signature* is a combination of a set of events. An *Event* is a signal with a particular value. The errata signature also has a field that specifies the maximum time interval within which we should detect all the events specified in the signature. The time interval can be specified either in terms of the number of committed instructions or processor cycles.

Figure 1 shows the high level architecture design for our hardware patching mechanism, which can be programmed using the errata signatures. The *Event-Signature Matcher* is programmed to watch for events (signals with specific values or values that lie within specified ranges) that are part of any of the errata signatures. When an event occurs, it can match a set of signatures in the Event-Signature Matcher. For each match, a $(SignatureID, EventID)$ tuple is generated and is given as input to the *Signature Monitor*.

The Signature Monitor contains an entry for each signature that is to be patched. The purpose of the Signature Monitor is to determine if all the events for a signature occur within the time interval specified for that signature. The signature monitor entry for a signature consists of the following: (1) a signature identifier (2) the time interval specified in signature (3) a list of events specified in the signature, and for each event it also keeps track of a timestamp. The timestamp maintained for an event indicates when that event last occurred. Thus, by examining the timestamps of all the events for a signature in the Signature Monitor entry, we can determine if all the required events have occurred within the specified interval.

When a $(SignatureID, EventID)$ tuple is generated by the Event-Signature Matcher, an entry in the Signature Matcher is updated as follows. The $SignatureID$ is looked up in the Signature Monitor table to select an entry. Then, the $EventID$ for that signature is updated with the current timestamp, to keep track of the time when the event last occurred. We then check to see if all of the events for the signature have occurred within the specified interval. If so, then we have detected an error corresponding to the monitored signature, and the signature identifier corresponding to the erratum is passed to the *Recovery* unit. The Recovery unit then initiates the recovery for the detected error using one of the mechanisms that we will describe in Section V.

In the remainder of this section, we discuss the types of signals that are important to monitor in order to detect the errata in Pentium4 and AMD64. Table III shows the common signal types involved in the processor errors. It also shows the percentage of errata that a particular type of signal is involved in. A particular error might require multiple signals. Hence, the numbers in the table do not add up to 100%. An important result in the table is that except for 22% of AMD64 errors and 31.2% of Pentium4 errors, the rest can be detected by monitoring a common set of signals. Hence, if the Event-Signature Matcher has access to signals of the types listed in Table III, it should be able to detect a majority of errors.

B. *Instruction Stream Signals*

Instruction stream signals are the ones routed to the micro-op expansion hardware. By having access to these signals, the detection unit can monitor the instructions that are getting executed. Instruction stream signals occur early enough during instruction execution. Therefore, the errors related to an instruction's implementation can usually be avoided at little cost, provided we can accurately identify the instances of the instructions that need to be fixed. From Table III, we see that about 25% of hardware errors in AMD64 and 27% of errors in Pentium4 require that the Event-Signature Matcher has the ability to monitor the instruction stream.

	AMD64	Pentium4
Instruction Stream	25.39	26.71
Exception	15.87	12.42
Interrupt	17.46	2.48
Mode Changes	11.11	19.88
Micro-Arch Events	19.04	13.66
Watchdog timer	6.34	1.86
Others	3.17	4.35
None required	22.20	31.2

TABLE III
SIGNALS INVOLVED IN ERRORS.

C. Interrupts and Exceptions

Interrupts are the asynchronous signals received by the processor from the external world. Defined exception signals are predicates indicating the ISA or implementation defined exceptions. We find that a significant proportion of errors in hardware require monitoring these events. Monitoring the interrupts and exceptions are useful in detecting 33.33% of AMD64 errors and 14.9% of Pentium4 errors.

Errata signatures with these events may not be seen early enough to avoid the corresponding erratum. However, rolling back the execution after flushing the pipeline or to a previous checkpoint [14] are possible approaches for correcting these errors. Also, we find that a significant proportion of errors involving exceptions are due to incorrect handling of the exception itself, which can be corrected using hypervisor support, which we will describe in Section V.

D. Monitoring the Mode Changes

We find that a large number of errors occur only while operating in a specialized mode. An example is executing in the low power mode or in the 32-bit compatibility mode. The Event-Signature Matcher that we propose should have access to the necessary signals to determine the mode, the processor is in. About 11.1% of errors in AMD64 and 19.9% of errors in Pentium4 occur while the processor is operating in a specialized mode.

E. Micro-architectural Events

In a processor, there are a set of signals that indicate various micro-architectural events. Performance counters in the modern processors are updated by monitoring such events. Examples in this group include L1 cache misses, TLB misses, hardware prefetches, etc. These signals need to be monitored to fix 19% of errors in AMD64 and 14% of errors in Pentium4 (see Table III).

F. Watchdog Timer

Finally, there is a class of errors that are not caught with the above monitors, which lead to a deadlock situation, where the processor stops making progress. This condition can be detected using a watchdog timer. Traditionally watchdog timers are used just to restart the processor after terminating the application that was running. We propose to instead use the watchdog timer to recover from the processor error with the help of hardware checkpoint mechanisms or hypervisor support, which we describe in Section V. We find that only about 6% of errors in AMD64 and 2% of errors in Pentium4 cannot be detected before the processor hangs.

G. Errors not Involving Signals

We found that for 22% of errors in AMD64, there is little use in using the signals to detect the hardware error. One reason is that there are errors that just cannot be patched by monitoring signals. For example, the erratum #104 in AMD64 is related to incorrect ECC calculation for partial writes. Another example is the erratum #111 in AMD 64 that says that the Rtt (Round-Trip-Time) specification for the HyperTransport pins used for communication is not meet. Then there are errors deemed as insignificant, which we place under this classification and will not be targeted for patching. An example is the erratum #75 in AMD64 which causes slight inaccuracy in the APIC timer.

H. Total Signals

As we will see in Section VII, we can cover 78% of the errors in AMD64 [5] by monitoring the signal types that we described above. We also estimated the total number of unique signals that need to be monitored to fix these errata. We found that we need to monitor around 41 signals to capture the 78% of errors in AMD64. The breakdown of the signals was as follows: 15 different signals were of type interrupts and exceptions, 14 different signals were related to micro-architectural events, 11 different signals were required to detect different modes in the processor, and one signal was required to monitor the opcodes of the instruction stream. This gives us an estimate of the number of signals that need to be made accessible to the Event-Signature Matcher.

V. HARDWARE DEFECT PATCHING SUPPORT

In Section IV, we discussed how one can detect the occurrence of an error by monitoring the signals in hardware. In this section, we describe various hardware mechanisms that can be used to patch the design errors when they manifest during program execution.

A. Instruction Stream Editing

The BIOS micro-code patching is useful to overcome those hardware errors that are due to incorrectly implemented instructions. However, these patches are static in nature, because they are applied once before the processor starts functioning. Given that the hardware errors in heavily tested processors occur very infrequently, it is inefficient to execute, potentially high overhead, patched micro-code sequence every time the instruction that may cause erroneous behavior is executed. The micro-code patch needs to be executed only when the processor is in a state that will cause an error if we do not execute the patch code.

Corliss et.al. proposed a dynamic instruction editing mechanism called DISE [9]. DISE is similar to the micro-code expansion mechanism in hardware, except that it is programmable and can inject instructions into the instruction stream when certain conditions are met. These conditions could be based on the opcode, the operand registers and their values, or they could even be based on the instructions that are currently being executed in the pipeline. We can use the instruction editing mechanism to execute patched micro-code sequences only when the required conditions are met.

B. Replay after Pipeline Flush

Many of the hardware errors cause problems under highly specific circumstances when two or more events occur close to each other in time. Using the errata signature, our hardware detector can detect the problem, and then to overcome the issue it might be sufficient to flush the pipeline and re-execute the instructions. Modern processors already have checkpoint support to roll-back to a mispredicted branch and replay from there. During replay, we can execute NOP instructions interspersed with the original instruction stream. Thereby, during replay, we can force sufficient delay between the events that would otherwise result in an erroneous execution. In Section VI-B, we discuss an example erratum that can be patched using this replay support.

C. Replay with Checkpoint Support

For the replay approach, we assumed that we have the capability to flush the instructions in the pipeline. But this replay functionality is inadequate to handle some errors that require replaying a few hundreds to thousand instructions across all the processors in a multi-processor system. For example, this is required in the case of hardware errors that are due to incorrect implementations of multi-processor functionalities like coherency, incorrect interaction with the memory subsystem, etc. One way to address these types of errors is to have a light-weight checkpoint support that can enable rolling back the architecture state a few hundred instructions for each of the cores in a multi-processor system. Then NOPs can be inserted into the instruction stream during the replay, to avoid the error.

Sorin et.al. proposed SafetyNet [14], which is a hardware assisted checkpoint mechanism that provides the capability to rollback program execution by about 100,000 cycles across all the processors in a multi-processor system. This type of checkpoint and restore could definitely be used to avoid some of the errors, but through our analysis of errata in Pentium4 and AMD64, we find that a light-weight mechanism, as mentioned above, which supports rolling back a few hundred instructions per core should be sufficient to recover a majority of complex hardware design errors.

1) Alternative to Checkpoint Scheme: If the above check-pointing schemes are not supported, then an alternative is to just detect the error and report it to the higher level software. This will allow the software to perform graceful recovery in the presence of an error. This way of handling the error is essentially how the watchdog timers are being used today. However, if the watchdog timer is used with the proposed Signature Monitoring unit, one can potentially report the cause for a deadlock when it is detected by the watchdog timer.

D. Hypervisor Patching Support

Some modern processors support hypervisors, also known as Virtualization Technology (VT). Recent examples of VT include Intel's Vanderpool [11] and AMD's Pacifica hardware [10]. A hypervisor occupies a layer between the operating system and the hardware. Because of its proximity to the hardware, patching using a hypervisor can cover a wider variety of problems without exposing the hardware error to the operating system or the software. In addition, VT hardware provides hypervisors with fine grained abilities to intercept and interrupt the flow of control when there is an exception. Hence, the proposed hardware error detector can trap to the hypervisor whenever an error is detected and communicate the erroneous condition to the hypervisor. The hypervisor can then take sophisticated corrective action and shepherd the program execution past the problem. This corrective action might involve flushing the pipeline or rolling back the execution to the previous checkpoint, and then replaying the program's execution.

VI. Case Studies

In this section, we discuss several case studies to illustrate how errors can be recovered using the patching mechanisms that we described in Section V.

A. Case for Instruction Stream Editing

The following errata can be found in the errata sheet for AMD64 [5]. When an `AAM` (ASCII Adjust after Multiply) is followed by another `AAM` instruction in a span of three instructions, or when a `DIV` is followed by an `AAM` in a span of six instructions, the processor might produce incorrect results. The suggested workaround for this problem in the errata sheet is to have the software ensure that the `AAM` and `DIV` instructions are sufficiently spaced out in the code using additional NOPs to avoid the error. Instead of exposing this problem to the software, we can use dynamic instruction stream editing described in Section V-A to fix this issue. The DISE [9] programmable engine can keep track of whether any `AAM` or `DIV` instruction was encountered in the last 6 instructions dispatched. If that condition is satisfied while dispatching an `AAM`, it can inject NOPs into the stream to avoid the impending hardware error. Note that, one could potentially use BIOS micro-code patching to solve this problem as well. However, that will involve injecting the NOPs for every instance of the `AAM` instruction, which can be expensive in terms of performance overhead.

B. Case for Replay after Pipeline Flush

The following errata found in the Intel Pentium4 errata sheet [4] can be solved with the help of replay support. Intel's Pentium processors have the support to do fast string copying operations while executing `MOVS` or `STOS` instructions with prefix `REP`. The processor makes use of the control register

CR2 while performing this string copy operation. If a paging event occurs while the processor is performing a fast string copy operation, then the value in the CR2 register can get incorrectly modified. In such circumstances, we will experience an incorrect execution of the program. There are no workarounds suggested in the errata sheet. However, this error can be easily detected using an errata signature that comprises of the following: 1) a paging event and 2) a fast string copy operation. The latter can be determined by monitoring the instruction stream signal to look for the opcodes with the prefix REP. The detection unit can detect this problem whenever the paging event is encountered and when the processor is in the fast string copy mode. After detection, the processor only needs to flush the pipeline and restart the execution from the last uncommitted instruction. Note, it is possible that the processor might take corrective action even when it is not absolutely necessary (false positive), but the approach guarantees that the error will not occur (no false negatives).

C. Case for Light-Weight Checkpoint Support

To overcome certain hardware errors, especially the errors related to the implementation of multi-processor functionalities, we might have to rollback past the committed instructions to undo the changes done to main memory. To rollback execution past the committed instructions in a multi-processor system, we assume a light-weight hardware checkpointing support that can support rolling back past a few hundred instructions in each processor in the system. An example is the following error reported in the Pentium4 [4] errata sheet (Errata R21): "While going through a sequence of locked operations, it is possible for the two threads to receive stale data. This is a violation of expected memory ordering rules and causes the application to hang". To recover from the deadlock, the execution can be rolled back to a past checkpointed state. During re-execution, we can ensure that the system does not encounter the same problem again by inserting NOPs into the instruction streams to cause sufficient delay between the lock operations.

D. Case for Hypervisor Support

In one of the AMD64 errata [5], Advanced Programmable Interrupt Controller (APIC) generated interrupt is not serviced correctly if the interrupt occurs while the processor is entering the C2 power state. The interrupt is not lost, but it is not serviced until some other wakeup event like a timer tick occurs to take the processor out of the C2 state. Unusually long delays in servicing the interrupt might result in unpredictable system failures. The suggested workaround for this bug in the errata sheet is to not enable the C2 power state. However, this amounts to loss of functionality in the processor. Instead, in our mechanism, since the detection unit has access to the interrupt signals and also can determine the current power mode in the processor, we can detect when such an interrupt occurs in the C2 power mode. Once detected, we can trap to the hypervisor where we can ensure that the interrupt is serviced without any delay.

In one other example from the AMD64 errata sheet [5], unexpected page faults are reported for software prefetches. But the right thing to do is to ignore the page faults triggered by the software prefetches. The suggested workaround to this problem is to modify the kernel of the operating system. Instead, our mechanism can handle this error as follows. The errata signature to detect the error is the combination of software prefetch instruction dispatch event and the page fault event. When the detection unit detects a match for the signature, it can trap to the hypervisor where we can accurately determine if the page fault was generated by the software prefetch instruction. If that is the case, then the fault can be ignored. If not, the hypervisor can invoke the page fault handler to service the page fault.

VII. Error Coverage Results

Table IV and Table V show the percentages of errors covered by the conventional approaches and the proposed mechanisms. The results are presented for the errors found in the errata sheets for AMD64 [5] and Pentium4 [4].

As shown in Table IV, using BIOS patches that do not involve disabling a functionality in the processor, we can patch 14.2% errors in AMD64 and 28.4% errors in Pentium4. A predominant proportion of these errors are the BIOS micro-code patches. However, executing the patch-up code can incur appreciable overhead. Instead of micro-code patches, we propose using *I-Stream* patching mechanism described in Section V-A, which can trigger the execution of patch-up micro-code conditionally and thereby avoid degradation in performance. From Table V, we can see that there are about 6.2% AMD64 errors and 10% Pentium4 errors that can benefit from conditional patching. However, 4.9% AMD64 errors and 9.1% Pentium4 errors still require patching every instance of an instruction (shown as *I-Stream:Static*).

There are about 12.6% AMD64 errors and 7.3% Pentium4 errors that require support from the higher level software. The categories labeled with the prefix *Disable* involve turning off some functionality in the processor (e.g., power saving feature or a prefetching mechanism). The conventional BIOS patching approach can be applied to 30% AMD64 errors and 10% Pentium4 errors, which disables certain features to avoid the hardware errors. If software support is used to ensure that certain features in the processor are not used or certain instruction sequences do not occur, then an additional 12.4% AMD64 errors and 17.3% Pentium4 errors can be covered. In total, about 42.4% AMD64 errors and 27.3% Pentium4 errors can be covered through these conventional techniques, but the downside is that many of the features will be disabled. We also found that about 25.3% AMD64 errors and 33.9% of Pentium4 errors cannot be patched using these conventional methods.

Table V shows the coverage using our signature-based detection and error correction mechanisms. Among our mechanisms, support for simple replay after flushing the pipeline can recover 7.9% AMD64 errors and 3.6% Pentium4 errors. Replay with a light-weight checkpoint support that we discussed in Section V-C can cover additional 9.5% AMD64

Technique	AMD64	Pentium4
BIOS	14.2	28.4
OS	6.3	5.5
Software	6.3	1.8
Disable:BIOS	30.1	10.0
Disable:OS	4.7	6.4
Disable:Software	6.2	3.6
Disable:External	1.5	7.3
Unimportant	3.1	2.7
Watchdog	1.5	0
Not covered	25.3	33.9

TABLE IV
COVERAGE USING CONVENTIONAL APPROACHES.

Technique	AMD64	Pentium4
I-Stream:Static	4.9	9.1
I-Stream:Cond	6.2	10.0
Replay w/ Pipeline Flush	7.9	3.6
Replay w/ Checkpoint	9.5	0.9
Hypervisor	46.0	39.4
Hypervisor+Replay	4.7	5.5
Disable:BIOS	7.9	0.0
Disable:Hypervisor	6.2	11.9
Unimportant	3.1	2.7
Not covered	4.7	16.5

TABLE V
COVERAGE USING SIGNATURE BASED PATCHING.

errors and 0.9% of Pentium4 errors. Hypervisor support plays an important role as it can help patching about 46% AMD64 errors and 39.4% Pentium4 errors. The category labeled as *Hypervisor+Replay* correspond to the technique where we have to flush the pipeline and start re-executing the program under the guidance of the hypervisor. The hypervisor will be in control of the program's execution until it has successfully shepherded the program's execution to get past the hardware error. The category labeled as *Disable:BIOS* and *Disable:Hypervisor* avoid the errors by disabling some functionalities in the processor with the help of BIOS or hypervisor support. The ones that are labeled as *Unimportant* correspond to errors like timer inaccuracy that do not affect the functionality of the processor. These errors do not warrant a patch.

In summary, using our proposed patching mechanisms, we are able to cover 78% of errors reported for the AMD64 processors and 69% of errors in the Pentium4 processor. When incorporating in the conventional disabling techniques, and ignoring the unimportant errors, we can cover almost all the errors except 4.7% AMD64 errors and 16.5% Pentium4 errors. Among the patched errors, only about 13% of AMD64 errors and 12% of Pentium4 errors require disabling some functionalities. This is better than 42.4% AMD64 errors and 27.3% that require disabling features using the conventional methods (many of which require patching support from software). Also, using the conventional methods 25.3% AMD64 errors and 33.9% of Pentium4 errors were left unpatched.

VIII. CONCLUSION

In spite of the enormous amount of effort spent on testing and validation, processors continue to contain non-trivial errors. We can only expect this situation to worsen as the hardware complexity continues to increase. Processor manufactures can benefit by including a mechanism in the processor that has the capability to patch the errors.

To patch an error, we require the ability to detect it and a mechanism to fix it. From our empirical study, we showed that it is possible to have a detection mechanism based on the errata signatures, as a majority of errors can be detected by monitoring a common set of signals. We also discussed using a set of architectural mechanisms to fix the error once they are detected when the processor is functioning. We showed that it is possible to cover 78% of errors reported for the AMD64 processors and 69% of errors in the Pentium4 processor using the proposed signature based error patching mechanisms.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for providing useful comments on this paper. This work was funded in part by Microsoft and Intel.

REFERENCES

[1] P. Rashinkara, P. Paterson, and L. Singh, *System-on-a-Chip Verification - Methodology and Techniques*. Kluwer Academic Publishers, 2001.
[2] AMD opteron woes. DailyTech. [Online]. Available: http://www.dailytech.com/article.aspx?newsid=2039&ref=y
[3] A. Wolfe. (1997, May) For Intel, it's a case of FPU all over again. EE Times.
[4] *Intel Pentium 4 Processor on 90nm Process, specification update*, Intel Corporation Std. Order No. 302 352-024, Dec 2005.
[5] *Revision Guide for AMD Athlon 64 and AMD Opteron Processors*, Advanced Micro Devices Std. Publication 25 759, Rev. 3.57, Aug 2005.
[6] M. Magee. (2002, Aug) Intel's hidden Xeon, Pentium 4 bugs. The Inquirer. [Online]. Available: http://www.theinquirer.net/default.aspx?article=5184
[7] L. C. Heller and M. S. Farrell, "Millicode in an IBM zSeries processor," *IBM Journal of Research and Development*, May 2004.
[8] T. M. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1999, pp. 196–207.
[9] M. L. Corliss, E. C. Lewis, and A. Roth, "DISE: A programmable macro engine for customizing applications," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*. New York, NY, USA: ACM Press, 2003, pp. 362–373.
[10] *AMD64 Virutalization Codenamed "Pacifica" Technology*, Secure Virtual Machine Architecture Reference Manual, Advanced Micro Devices Std. 33 047, Rev. 3.01, May 2005.
[11] (2005, Dec) Intel virtualization technology. Intel Corporation. [Online]. Available: http://www.intel.com/technology/computing/vptech/
[12] A. Klaiber, "The technology behind the crusoe processors," White paper, Transmeta Corporation, Jan 2000.
[13] A. Avizienis and Y. He, "Microprocessor entomology: A taxonomy of design faults in COTS microprocessors," in *Proceedings of the Conference on Dependable Computing for Critical Applications (DCCA)*. Washington, DC, USA: IEEE Computer Society, 1999, p. 3.
[14] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood, "SafetyNet: Improving the availability of shared-memory multiprocessors with global checkpoint/recovery," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*. IEEE Computer Society, 2002, pp. 123–134.