Patching Processor Design Errors with Programmable Hardware

Smruti Sarangi
University of Illinois, Urbana-Champaign

Satish Narayanasamy
Bruce Carneal
University of California, San Diego

Abhishek Tiwari
University of Illinois, Urbana-Champaign

Brad Calder
University of California, San Diego, and Microsoft

Josep Torrellas
University of Illinois, Urbana-Champaign

Equipping processors with programmable hardware to patch design errors lets manufacturers release regular hardware patches, avoiding costly chip recalls and potentially speeding time to market. For each error detected, the manufacturer creates a fingerprint, which the customer uses to program the hardware. The hardware watches for error conditions; when they arise, it takes action to avoid the error.

Today's processors are so complex that design validation and testing is a major bottleneck, often accounting for 50 to 70 percent of processor development time.[1] Yet, despite all the labor and computing resources invested in verification, many defects still slip into production silicon. At that point, dealing with them can be very costly. Specifically, if a manufacturer finds serious errors before the chip is released, the costs involve the engineering time required to debug and fix the errors; the direct fabrication costs associated with a chip respin, including new masks; and, especially, the lost revenue attributable to delayed shipping. If errors surface after the chip's release, the company's costs include all of the above plus the costs of performance-impairing or software-intensive workarounds, potential security breaches, chip recalls, and a tarnished reputation.

Perhaps the most publicized design error to date was the floating-point division bug in the Pentium processor, which led to a $475-million chip recall.[2] In 1999, a design error in the Pentium III temporarily halted shipment of Intel servers. Problems in the Pentium 4's cache and prefetch engine temporarily led to disabling prefetching in multiprocessor systems. More recently, design errors led to a recall of Itanium 2 processors, incorrect results in the AMD Athlon 64, and circuit errors that prevented the IBM PPC 750GX from running at 1 GHz. In fact, almost every modern processor has tens or even hundreds of design errors, which manufacturers discover after shipment and publish in errata sheets. (See the "Errata sheets" sidebar for details on these publications.)

To date, most of the resources committed to preventing processor design errors have gone to validation and testing. The fact that many errors are still getting through suggests that novel approaches are necessary to better handle errors. The approach that we propose in this article enables manufac-

Published by the IEEE Computer Society

turers to treat design errors like software bugs. Specifically, upon discovering a new error in a processor, the chip vendor creates a hardware patch that customers can apply to the chips in the field. Once installed, the patch automatically averts or repairs the error on the fly.

Providing processor support for patching design errors has several benefits. First, it allows a company to avoid expensive chip recalls and the accompanying damage to its reputation. Second, for errors that would not lead to recalls, patches obviate workarounds that disable functionality or impair performance. Finally, using this approach could enable a processor manufacturer to potentially save the last 10 weeks of testing, during which the detection rate of new bugs drops nearly to zero,[1] and thereby market the chip earlier and gain a valuable edge over competitors.

Our proposal is to include a field-programmable hardware mechanism in the processor, which can be used to patch errors.[3,4] On discovering an error, a processor manufacturer composes what we call an *error fingerprint*, which consists of a set of error conditions and a time interval. The proposed hardware can be programmed using the error fingerprint to monitor the necessary signals in the processor, looking out for the error conditions. If all the required error conditions occur within the specified time interval, the mechanism flags an error and initiates an appropriate recovery.

For this patching mechanism to be feasible, the hardware must be capable of monitoring all the signals necessary to determine the error conditions. A key observation makes this feasible: Our analysis of the errors in modern processors shows that programming the proposed hardware with a well-defined set of signals will detect a majority of errors, often before the errors have corrupted the system. We also show that it is possible to determine the required set of signals, and the size of the programmable hardware, a priori during processor design, by studying errors in earlier designs.

To recover from the errors detected by our programmable error detector, we examine mechanisms that include using instruction-stream editing,[5] flushing the pipeline, rolling back execution, and providing hypervisor support. By empirically analyzing the design errors in 15 different processors, we have found that these mechanisms can patch a large portion of errors. For example, our mechanisms can patch 78 percent of AMD64 errors and 69 percent of Pentium 4 errors without degrading functionality or performance. Finally, the hardware we propose incurs negligible area and wire overhead.

Errata sheets

After releasing a processor, the manufacturer typically publishes an errata sheet that details the processor's design errors and suggests possible hardware and software workarounds. Following are several errata sheet examples. The first two in the list are the Intel Pentium 4 and the AMD Athlon 64 and Opteron errata sheets from which we drew the error and workaround information we discuss in this article.

1. *Intel Pentium 4 Processor on 90nm Process,* Intel Std. Order No. 302 352-024, Intel, 2005.
2. *Revision Guide for AMD Athlon 64 and AMD Opteron Processors,* AMD Std. Publication 25 759, Rev. 3.57, Advanced Micro Devices, 2005.
3. *Intel Itanium 2 Specification Update,* Intel Std. Document No. 251 141-030, Intel, 2005.
4. *IBM 750FX Technical Documentation,* IBM Std. Document No. DD2.X, IBM, 2004.
5. *Freescale MPC7457CE Technical Documentation,* Freescale Std. Revision 10, Freescale, 2004.

Current mechanisms for patching errors

Certain processor design errors can be patched using mechanisms that already exist in processors. We'll look first at these mechanisms and explain their limitations.

Firmware patches

Processor vendors often apply hardware workarounds by modifying the BIOS or board settings. Changes at this level are convenient, because they can use full knowledge of the system's configuration and do not affect higher-level software users. The BIOS and board settings are used to set up the initial control state to circumvent the error. The changes typically involve disabling a certain feature (such as prefetching support), limiting the frequency or voltage, or connecting pins to certain logic values. For example, it's possible to patch a design error in the Pentium 4 processor by using the BIOS to disable cache prefetching. Setting the control bits appropriately to

increase the signals' pulse width overcomes erratum 98 in the AMD64 errata sheet (see the "Errata sheets" sidebar).

A related common hardware technique is the use of microcode patches or opcode traps. Modern CISC processors, such as the x86 series, have a microcode translator that translates complex CISC instructions into a sequence of micro-operations. Therefore, if an error is related to the implementation of an instruction with a particular opcode, we can patch it by simply changing the microcode translation corresponding to that opcode.

Some processors have a firmware layer above the processor that performs complex processor functions. For example, Itanium's firmware performs functions related to translation look-aside buffer (TLB) and floating-point unit control, and interrupt and error-correcting code (ECC) handling (see http://www.intel.com/design/itanium/firmware.htm). IBM's zSeries machines use Millicode, which, in addition to implementing complex instructions, also allows control of the processor's state.[6] Thus, IBM can use Millicode to patch errors related to a particular instruction's implementation and also to set the control states to disable features that lead to error conditions. Finally, Crusoe has placed, above the processor, a layer of code-morphing software, which translates the x86 instructions into the native ISA (see http://www.transmeta.com/pdfs/paper_aklaiber_19jan00.pdf). This support offers opportunities to patch errors related to the execution of certain instruction opcodes.

However, these current technologies are not very suitable for the types of complex error conditions we examine in this article. They suffer from three main disadvantages: First, a significant proportion of the BIOS and other firmware patches that processor manufacturers suggest involve disabling a processor feature. Such patches can degrade the processor's performance or functionality. Examples include disabling the write-combining feature in write buffers (AMD64 erratum 133) and disabling a power optimization feature (AMD64 erratum 78). Second, current mechanisms use a very inefficient approach to handle the complex errors we consider here. Each of our errors occurs when there is a subtle combination of events; such an occurrence does not correlate well with the execution of any particular instruction opcode. Performing additional checks or trapping at each instance of an opcode's execution would substantially hurt performance. For example, erratum 103 for AMD64 says that incorrect execution of the AAM instruction occurs only under a very specific pipeline condition. A possible microcode patch would involve executing additional NOPs (no operations). Executing this inefficient microcode for every dynamic instance of the AAM instruction would degrade performance. Finally, for many errors, it is unclear how to construct a patch using only microcode.

Software patches

Errata documents also suggest software workarounds that involve changes to the compiler or operating system. For example, to avoid a particular sequence of events that triggers an error, code generation in a compiler can make sure that certain opcodes never appear adjacent to each other. Similarly, the operating system can limit the order of initializing devices or processing interrupts, or limit the range of settings allowed for some parameters. However, software workarounds typically reduce performance or functionality in some way. Moreover, they are mostly effective only if a single operating system and compiler or linker is used for the processor; otherwise, they require a different version for each different operating system and compiler that the processor supports. 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 DEC was in complete control of the operating system (OSF) and the compiler chain.

Respin

Among post-fabrication patching options, respinning the chip is the most

effective, but it is also the costliest. Although respinning can fix errors, it can also introduce new errors. Respin is expensive because it requires additional engineering time, the purchase of new masks, a potentially devastating delay to market, and even the replacement of customer products. Although respinning can completely fix a hardware error, it is far more costly and time-consuming than sending a hardware patch to the customer.

Checker for pipeline core

An alternative approach is to include a well-tested in-order checker processor such as DIVA.[7] The checker processor redundantly executes the instructions to verify the correctness of a complex, out-of-order pipeline. However, this approach is not very effective for the design errors we are considering. First, DIVA focuses mostly on checking the pipeline, while most errors we consider are in the core's periphery. Second, many errors are caused by side effects, such as cache corruption; simply reexecuting the instruction does not help. Finally, instructions with I/O or multiprocessor effects cannot simply be reexecuted.

Concurrently with our work, Wagner et al. proposed using a content-addressable memory (CAM) that can be programmed to detect the errors in a processor's control logic.[8] The detection mechanism works by detecting illegal control states and transitions. Their proposed recovery technique consists of flushing the pipeline and reexecuting in a formally verified safe mode. Because the detection mechanism mostly monitors the state transitions in the control logic of the pipeline's core, their technique is mainly limited to patching design errors there. In contrast, our mechanism can detect and recover from more complex design errors involving error conditions even at the periphery of the processor. We achieve this by using a distributed patching mechanism that can correlate error conditions occurring in different parts of a processor over a period of time.

Processor design errors

To gain insight into the nature of design errors and their areas of concentration in modern processors, we studied the errata sheets for several processors.

This article combines and summarizes two earlier works. We focus here on the design errors in AMD Athlon 64 and AMD Opteron processors (AMD64) and Pentium 4 processors in one of the articles.[3] Analysis and results for 13 additional processors, including Intel Pentium III, Itanium 1, Itanium 2 and Pentium M, AMD K6 and Athlon, IBM PowerPC G3, Motorola G4, Sun Ultra Sparc II, and several network and embedded processors can be found in the other article.[4]

Source of design errors

The AMD64 errata sheet reports 63 errors; the Intel Pentium 4 errata sheet reports 109 (see the "Errata sheets" sidebar). Table 1 classifies these design errors in four broad classes: new features, external events, peripheral features, and miscellaneous. Each of the four classes contains additional specific subclasses. A few of the errors fall into more than one subclass, so the percentages for a processor do not add up to 100. For clarity, Table 1 also contains the absolute number of errors in each subclass.

New features. New features added to a processor design are a significant source of design errors. The first two rows of Table 1 list the errors related to Intel's Hyper-Threading and Vanderpool technologies, which respectively resulted in 12 and 9 of the Pentium 4's total 109 errors. (We did not find any errors in the AMD64 sheet related to the implementation of Virtualization technology.)

Power management is a relatively new functionality added to these CPU designs. This subclass contains 11.1 percent of the errors in AMD64 and 3.7 percent of the errors in Pentium 4. Recently added 64-bit extensions account for 9.5 percent of the errors in AMD64 and 11.0 percent of the errors in Pentium 4. Interesting examples in this error type include various incorrect implementations of the CISC-based string instructions when they operate on operands that are over 2^{32} bits long.

Table 1. Classification of design errors in AMD64 and Intel Pentium 4.

		Errors in AMD64		Errors in Pentium 4	
Class	**Subclass**	**Percentage**	**No.**	**Percentage**	**No.**
New features	Hyper-Threading	N/A	N/A	11.0	12
	VT (Vanderpool)	N/A	N/A	8.3	9
	64-bit extension	9.5	6	11.0	12
	Power management	11.1	7	3.7	4
External events	Interrupts	7.9	5	1.8	2
	Memory interface	17.4	11	20.2	22
	Multiprocessor	14.2	9	5.5	6
Peripheral features	Incorrect error report	11.1	7	6.4	7
	Debugging support	3.1	2	11.0	12
	Exception	6.3	4	2.7	3
Miscellaneous	Pipeline core	11.1	7	16.5	18
	Frequency/electrical	14.2	9	4.5	5
	Others	3.1	2	0.9	1

External events. Interrupts are used for interfacing the processor with the external devices. We found five errors in AMD64 and two errors in Pentium 4 that were directly related to the implementation of interrupt handling. The memory interface subclass includes the errors in the bus interfaces, cache, and virtual memory implementations. About 17.4 percent of errata in AMD64 and 20.2 percent in Pentium 4 fall into this subclass, making it the largest. All the errors that relate to the interactions between multiple processors (for example, coherence-related errors) fall into the multiprocessor subclass.

Peripheral features. Processors incorporate diagnosis functionality to detect certain faults—for example, ECC serves to detect memory faults. We found seven errors in both AMD64 and Pentium 4 that stemmed from incorrect error reporting. Incorrect error reporting is a recurring problem in these implementations, but the impact of these errors is usually less than catastrophic because they do not lead to incorrect program execution. Errors in this subclass include off-by-one counters, mismanaged counter overflows, and extended delays when reporting a condition. The next subclass relates to hardware debugging support; a substantial number of these errors had to do with altered execution flow of the application. Various problems involving the single-step execution facility fall into this subclass. Mismanagement of the data watch-point capability is also common. Incorrect handling of internal (CPU-originating) exceptions fall into the peripheral features class as well.

Miscellaneous. The first subclass in this category consists of errors due purely to errors in the pipeline core that lead to incorrect execution of instructions with certain opcodes. Thus, we've found that only about 11.1 percent of errors in AMD64 and 16.5 percent of errors in Pentium 4 can be fixed by verifying the pipeline's function, which is feasible with techniques such as DIVA. Most other errors are due to new or peripheral features, or due to functionalities that support complex interactions with the external devices.

The frequency/electrical subclass represents all errors that occur only with specific operating frequencies or clock ratios, as well as errors that arise when the processor does not meet certain electrical specifications (such as the voltage specification). Several errors in this subclass occur in conjunction with peculiar but legal motherboard configuration choices.

Table 2. Importance of design errors in AMD64 and Intel Pentium 4 processors.

Error type	AMD64 (%)	Pentium 4 (%)
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

Importance of design errors

Two factors determine the importance of an error: frequency of occurrence and severity. Unfortunately, the errata documents available to us give only a limited view of the frequency of the individual errors. However, they do tell us whether an error can occur at the customer site or only when the processor undergoes contrived, in-house testing. Table 2 lists the percentages of customer and in-house errors for AMD64 and Pentium 4 processors.

Clearly, errors that have the potential to occur at the customer site are important. However, for certain errors, such as a very small time slip in the performance counter accounting, the errata sheets indicate that they are not a cause for concern. Therefore, we classify customer errors as either important or unimportant.

For errors found during in-house testing, the manufacturers indicate whether or not they plan to fix them, so we further classify in-house errors on that basis. We consider errors that the manufacturer plans to fix important. Thus, adding the first and the third rows in Table 2, we see that approximately 90 percent of errors are important. A mechanism to patch these errors in shipped processors will be of great value.

Programmable hardware for detecting errors

We propose that processors include a programmable hardware patching mechanism to repair errors in the field. Upon discovering an error, the manufacturer composes an error fingerprint and distributes it to customers, who can use it to program the on-chip programmable patching hardware.

Error fingerprint

An error fingerprint is used to program the proposed hardware patching mechanism to dynamically detect and recover from a design error. It specifies a set of conditions and a time interval within which those conditions must occur for an error to be flagged. The fingerprint can specify the time interval in terms of either the number of committed instructions or the number of processor cycles. Another alternative is to specify the time interval using a starting and an ending condition.

The ''Processor design error examples'' sidebar describes several errors to which we will refer in describing our mechanism. For Error 2 in the sidebar, the error fingerprint consists of two conditions and a time interval. The conditions are

- execution of MOVS or STOS instructions with prefix REP, and
- a page fault exception.

The dispatch and commit of the MOVS or STOS instruction with prefix REP are the starting and ending conditions respectively, which together specify the time interval for the error fingerprint.

Patching architecture overview

Figure 1 shows a high-level view of the proposed hardware patching mechanism. Logically, it consists of three units: the condition detector, the error detector, and the recovery unit. The *condition detector* contains an entry for each condition to be detected during the processor's execution. It is designed so that it can monitor a set of signals necessary for detecting most of the error conditions. We can broadly group the input signals required to detect the condi-

Processor design error examples

- *Error 1.* In AMD64, when an AAM (ASCII adjust after multiply) instruction is followed by another AAM instruction within a span of three instructions, or when a DIV is followed by an AAM within a span of six instructions, the processor can produce incorrect results.
- *Error 2.* Intel's Pentium 4 processor supports fast string-copying operations while executing MOVS or STOS instructions with prefix REP. The processor uses control register CR2 while performing this string-copy operation. If a paging event occurs while the processor is performing a fast string-copy operation, the value in the CR2 register can be incorrectly modified. These circumstances result in incorrect program execution.
- *Error 3.* In Pentium 4, 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.
- *Error 4.* In AMD64, unexpected page faults are reported for software prefetches.

tions into those coming from the pipeline core (such as branch misprediction and opcodes of instructions dispatched), memory subsystem (such as cache misses), I/O and interrupt (such as watchdog timer interrupts and other exceptions), and system functions (such as signals that indicate power and other mode changes). To patch an error, the customer uses the error fingerprint to program the condition detector so that it looks out for the necessary error conditions during processor execution.

When the condition detector detects an error condition, it communicates the detected condition to the *error detector,* whose purpose is to determine whether all the error conditions for a fingerprint have occurred within the specified time interval. The error detector keeps track of the following information for an error fingerprint:

- fingerprint identifier,
- list of error conditions specified in the fingerprint, and
- time stamp for each error condition.

The time stamp maintained for an error condition indicates when it last occurred. Thus, by examining the time stamps of all the error conditions for a fingerprint, the error detector determines whether all the required error conditions have occurred within the specified time interval. If so, the error detector flags an error and informs the *recovery unit.* If the time interval in the error fingerprint is specified using starting and ending conditions, then all the error conditions for that particular error fingerprint are cleared both when the starting and the ending conditions are satisfied. The recovery unit initiates one of several recovery mechanisms that we describe later.

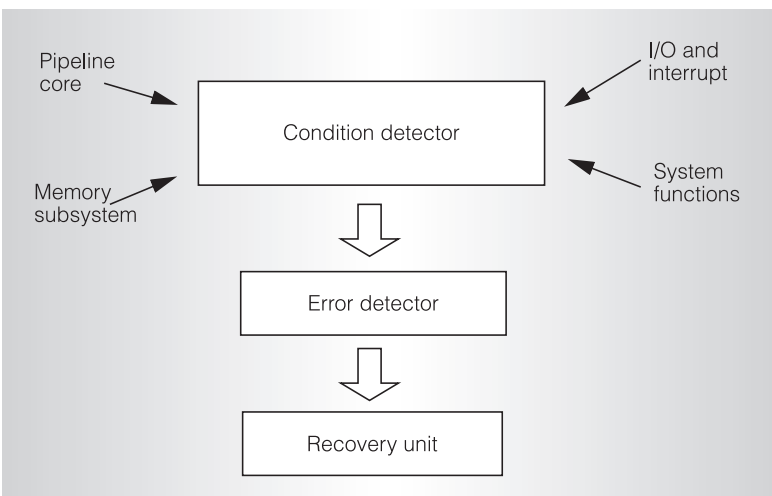

Figure 1. High-level view of the architecture for patching design errors.

Distributed design for programmable patching hardware

It is impractical to route all the necessary signals to a centralized condition detector and to communicate the detected conditions to a centralized error detector. Hence, we propose a distributed design for these two units, as Figure 2 shows.

We logically partition the processor into a few subsystems—such as the fetch unit, the data cache, and the memory controller—and assign one condition detector and one error detector to each subsystem. An individual condition detector monitors signals from the subsystem it is a part of. An individual error detector receives most of its input signals from the condition detector in its subsystem, but it also receives signals from other condition detectors. Such signals travel over cross-subsystem wires that go through programmable interconnect modules. These programmable interconnect modules are judiciously distributed on-chip in different neighborhoods of subsystems and are automatically programmed in the field using the information from the error fingerprint.

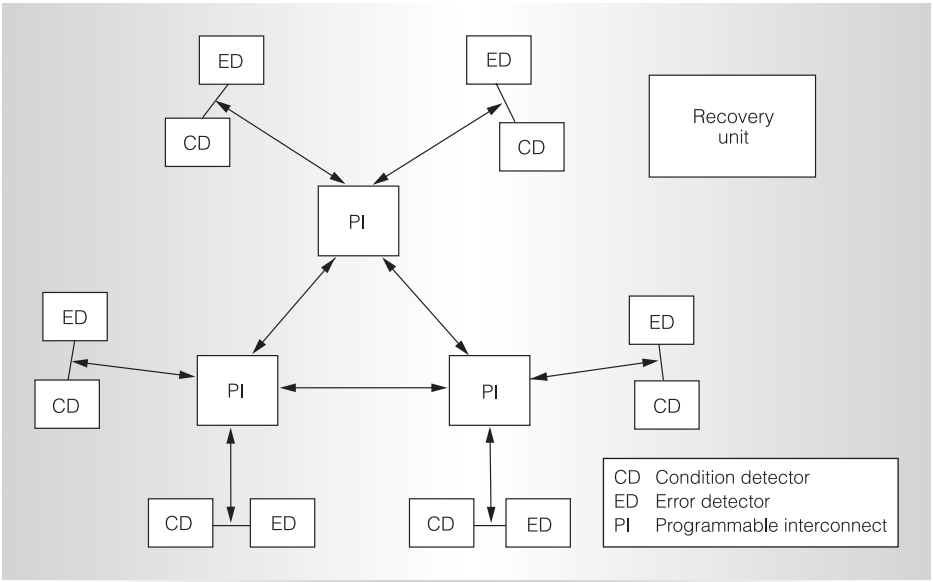

Figure 2. Distributed design of our programmable error-patching hardware.

For certain errors, a single subsystem generates all the required conditions. Such errors are detected by the condition detector in that subsystem. For other errors, condition detectors in different subsystems detect the required conditions and route them to a single error detector. In both cases, as soon as an error detector detects an error, it informs the global recovery unit, which then initiates recovery.

Recovery support

For the recovery unit, we evaluated four different types of mechanisms: instruction-stream editing, replay after pipeline flush, replay with checkpoint support, and hypervisor patching support.

Instruction-stream editing

BIOS microcode patching is useful for overcoming hardware errors due to incorrectly implemented instructions. However, these patches are static in that they are applied once before the processor starts functioning. Thus, the processor must execute the potentially high-overhead, patched-up microcode sequence every time when it executes the erroneous instruction, possibly degrading performance. In our system, the patched code is executed only under the rare circumstances when the error detector detects an error condition that can result in an incorrect execution.

Corliss et al. proposed a dynamic instruction-stream (I-stream) editing mechanism called DISE, which is similar to the microcode expansion mechanism in hardware, except that it is programmable and can inject instructions into the instruction stream only under certain conditions.[5] These conditions could be based on the opcode, the operand registers and their values, or even the instructions currently being executed in the pipeline. We can use the instruction-editing mechanism to execute patched microcode sequences only under the required conditions.

For Error 1 in the "Processor design error examples" sidebar, the suggested workaround 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 fix it using dynamic I-stream editing. Our architecture would keep track of whether any AAM was one of the last five instructions dispatched. If that condition occurs during dispatch of an AAM, DISE would inject NOPs into the stream to avoid the impending hardware error. One could use BIOS microcode

patching to solve this problem, but that would involve injecting the NOPs for every instance of the AAM instruction, and thus degrade performance.

Replay after pipeline flush

Many of the errors we consider can be detected before they corrupt the architectural state. Also, in a heavily tested processor, an error occurs only under a highly specific circumstance, when two or more conditions occur close to one another in time. Thus, when the error detectors catch such a circumstance, for many errors it is sufficient just to flush the pipeline and replay from the last committed instruction. Modern processors already have this replay capability to handle branch mispredictions.

If a simple replay is not sufficient to avoid the error conditions, we can change the event interleaving during replay through several means. One option is to inject NOPs into the instruction stream during replay. This forces delay between the conditions that would together result in an erroneous execution. We can also disable one of the problem-causing signals during replay, such as the low-power mode, and enable it again after getting past the error conditions. Another option is to generate an alternative sequence of micro-operations for certain instructions during replay. Finally, we could use hypervisor support to emulate the instructions until execution gets past the error conditions.

We can detect Error 2 (listed in the sidebar) by programming the proposed hardware with the error fingerprint we described earlier. Upon detection, the processor need only flush the pipeline and restart execution from the last committed instruction, because the paging event would already have been handled and is unlikely to occur again during replay. It is possible for the patching hardware to take corrective action even when it is not absolutely necessary (that is, false-positive triggers for recovery are possible), but the mechanism guarantees that the error will not go undetected (that is, there can be no false negatives).

Replay with checkpoint support

The pipeline flush and instruction replay functionality is inadequate to handle certain errors that can corrupt the cache state before error detection. One way to address these types of errors is for the processor to support lightweight checkpoint and rollback of the cache state. Such checkpoint support is similar to that required for thread-level speculation or transactions.

As we observed earlier, some errors pollute the memory or even the I/O state. Some of these errors are related to incorrect implementations of multiprocessor functionality such as cache coherence. Recovery from these errors requires heavier-weight schemes that support memory or I/O state rollback and replay—possibly over many hundreds of instructions for each of the cores in a multiprocessor system. SafetyNet and ReVive support low-overhead checkpoint and rollback of memory state.[9,10] ReViveI/O enables I/O undo and redo.[11]

However, our analysis indicates that a lightweight checkpoint mechanism that would allow rolling back cache states should be sufficient for recovery from a majority of complicated hardware design errors. For Error 3 in the sidebar, to recover from the deadlock, the execution can be rolled back to a past checkpointed state. During reexecution, we can ensure that the system does not reencounter the problem by inserting NOPs into the instruction streams to cause sufficient delay between lock operations.

If the processor does not support the checkpointing schemes just described, an alternative is just to 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 method of handling errors is essentially how watchdog timers function today. However, if a processor uses a watchdog timer along with our proposed architecture, our hardware could potentially report the cause for a deadlock when the watchdog timer detects it.

Hypervisor patching support

Some modern processors support *hypervisors,* also known as *virtualization technol-*

ogy (VT). Recent examples of VT include Intel's Vanderpool (http://www.intel.com/technology/virtualization/index.htm) and AMD Virtualization (http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_8826_14287,00.html). 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, hypervisors provide fine-grained capabilities to intercept and interrupt the control flow when there is an exception. Hence, for certain errors, upon detection, the recovery unit can trap to the hypervisor and communicate the erroneous condition to it. 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 execution to the previous checkpoint and then replaying.

For Error 4 in the sidebar, page faults triggered by the software prefetches should be ignored. The suggested workaround for this problem is to modify the operating system's kernel. Instead, we can detect the error by programming the patching hardware using an error fingerprint that consists of two conditions: a software prefetch instruction dispatch condition and a page fault condition. Upon detection, the recovery unit traps to the hypervisor, which then accurately determines whether the page fault was generated by the software prefetch instruction. If so, the processor can ignore the fault. If not, the hypervisor can invoke the page fault handler to service the page fault.

Results

To assess the proposed hardware patching mechanism, we analyzed its area, wire, and performance overhead. We also compared the number of errors that can be patched using our mechanism versus the current possible workarounds. Finally, we studied methods of sizing and designing the proposed hardware mechanism during development of a new processor using training data from previous processors.

Overhead

To determine the number of signals that would have to be made accessible to the condition detectors, we account for all the signals listed in the errata sheets that are part of at least one error, including both Generic and Specific signals. Generic signals are microarchitectural signals largely common to all processors, such as cache miss, bus transaction, or interrupt. Specific signals are specific to a processor, such as special pins or registers in the processor. Table 3 lists the number of signals required, classified into four broad classes of subsystems. In total, AMD64 requires 185 signals, and Pentium 4 requires 270 signals. Of these, about 150 signals are Generic, and the rest are Specific.

Based on the number of signals watched, we sized the architectural structures in the distributed design described earlier; the area and wiring overheads are negligible.[4] Specifically, using data from Khatri et al.,[12] we estimate that the area overhead is on the order of 0.06 percent for Pentium 4 and 0.03 percent for AMD64. Moreover, using Rent's rule, we estimate that the increase in the number of wires is about 0.29 percent for AMD64 and 0.86 percent for Pentium 4.

Our hardware patching mechanism also has negligible execution overhead. Performance is adversely affected during the recovery operation; however, because the error rate is very low, the performance impact due to recovery is very small. Moreover, although tapping signals increases wire load, an actual implementation is likely to affect performance negligibly.

Coverage results

Tables 4 and 5 show the percentages of errors covered by the conventional approaches and the proposed mechanisms, for the errors from the errata sheets for AMD64 and Pentium 4.

Coverage for current workarounds. As Table 4 shows, using BIOS patches that do not involve disabling any processor functionality, we can patch 14.2 percent of errors in AMD64 and 28.4 percent of errors in Pentium 4. A predominant proportion of these patches are the BIOS microcode

Table 3. Signals monitored by condition detectors.

	No. of signals monitored	
Subsystem	**AMD64**	**Pentium 4**
Pipeline core	40	50
Memory subsystem	90	110
I/O and interrupt	25	50
System functions	30	60
Total	185	270

patches. However, executing the patched microcode for every instance of the erroneous instruction can incur appreciable performance overhead. Table 4 also shows that about 12.6 percent of AMD64 errors and 7.3 percent of Pentium 4 errors require support from the higher-level software.

The categories labeled with the *disable* prefix involve turning off some feature in the processor (for example, a power-saving feature or a prefetching mechanism). The conventional BIOS-patching approach can work for 30 percent of AMD64 errors and 10 percent of Pentium 4 errors, when certain features are disabled to patch the hardware errors. In addition, by using software support and by reconfiguring the external hardware states, certain features in the processor can be disabled or certain instruction sequences can be avoided to patch an additional 12.4 percent of AMD64 errors and 17.3 percent of Pentium 4 errors (sum of *disable-OS, disable-software,* and *disable-external* in Table 4).

In total, conventional workarounds can patch about 42.4 percent of AMD64 errors and 27.3 percent of Pentium 4 errors. The downside is that many of the patches involve disabling features. In addition, about 25.3 percent of AMD64 errors and 33.9 percent of Pentium 4 errors cannot be patched using these conventional workarounds.

Coverage for proposed patching hardware. Table 5 shows the coverage for our error-fingerprint-based patching hardware. The I-stream editing mechanism triggers the execution of patch-up microcode only when the required error conditions are satisfied. Thus, it avoids the performance inefficiency of BIOS microcode patching. About 6.2 percent of AMD64 errors and 10 percent of Pentium 4 errors can benefit from condi-

Table 4. Error coverage for conventional workarounds.

	Errors covered (%)	
Technique	**AMD64**	**Pentium 4**
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 5. Error coverage for error fingerprint-based hardware patching mechanism.

	Errors covered (%)	
Technique	**AMD64**	**Pentium4**
I-stream, static	4.9	9.1
I-stream, conditional	6.2	10.0
Replay with pipeline flush	7.9	3.6
Replay with checkpoint	9.5	0.9
Hypervisor	46.0	39.4
Hypervisor plus 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

tional patching. However, 4.9 percent of AMD64 errors and 9.1 percent of Pentium 4 errors still require executing the patch-up microcode for every instance of the erroneous instruction (*I-stream, static* in Table 5).

Among our recovery mechanisms, support for simple replay after flushing the pipeline can recover 7.9 percent of AMD64 errors and 3.6 percent of Pentium 4 errors. Replay with a lightweight checkpoint support can cover an additional 9.5 percent of AMD64 errors and 0.9 percent of Pentium 4 errors. Hypervisor support plays an important role in recovery; it can patch about 46 percent of AMD64 errors and 39.4 percent of Pentium 4 errors. *Hypervisor plus replay* corresponds to the technique that flushes the pipeline and starts reexecution of the program under the guidance of the hypervisor. The hypervisor maintains control of the program execution until it has successfully shepherded the program's execution past the hardware error. *Disable-BIOS* and *disable-hypervisor* avoid the errors by disabling some functionalities in the processor with the help of BIOS or hypervisor support. The *unimportant* category corresponds to errors such as timer inaccuracy that do not affect processor functionality; these errors do not warrant a patch.

In summary, using our proposed patching mechanisms, we can cover 78 percent of errors reported for the AMD64 processors and 69 percent of errors for the Pentium 4. If we also consider conventional disabling techniques and ignore unimportant errors, we can cover all but 4.7 percent of AMD64 errors and all but 16.5 percent of Pentium 4 errors. Among the errors patched this way, only about 13 percent of AMD64 errors and 12 percent of Pentium 4 errors require disabling a processor feature. This is better than the conventional methods, which require disabling features for 42.4 percent of AMD64 errors and 27.3 percent of Pentium 4 errors; many of these errors also require patching support from software.

Designing the hardware patcher for new processors

As engineers design a new processor, they do not know what design errors it will have and, therefore, do not know how to size and lay out the patching hardware of Figure 2. To solve this problem, we propose an algorithm that, given the Specific signals in the processor (obtained from the processor's manual) and a rough layout of its subsystems on the chip, can size and lay out the patching hardware—signals tapped, condition detectors, error detectors, programmable interconnects, and wires.[4] We obtain the rules in this algorithm by generating scatter plots of the hardware needed from a training set of older processors.

We also used this algorithm to predict the hardware needed for several new processors. We find that training the rules of the algorithm with seven processors is accurate enough to size the patching

hardware of new processors to cover more than 95 percent of the design errors that would be covered with perfect knowledge. In addition, the processors in the training set do not have to resemble the new processors.

Today's processors have many nontrivial errors, even though companies spend enormous resources validating and testing the designs. We can only expect this problem to become more severe as processor complexity increases: Increased integration will increase verification effort while hurting signal observability; larger design teams will increase the risk of errors due to miscommunication; finally, more sophisticated features—and the ambiguities of the many new standards supporting them—will also contribute to errors. Our proposed programmable hardware mechanism for patching errors offers processor manufacturers a new and better way to deal with these inevitable errors. Manufacturers can regularly release hardware patches to fix errors, avoiding costly chip recalls and respins, and potentially releasing silicon to market earlier. Overall, our scheme enables an exciting new environment where hardware design errors can be handled as easily as system software bugs, by applying a patch to the hardware. MICRO

Acknowledgments

We thank T.N. Vijaykumar for his helpful comments on this article. This work was funded in part by NSF under grant CCR-0325603; by DARPA under grant NBCH30390004; and by Microsoft, Intel, and IBM.

References

1. A. Gluska, ''Coverage-Oriented Verification of Banias,'' *Proc. Design Automation Conf* (DAC 03), ACM Press, 2003, pp. 280-285.
2. T.R. Halfhill, ''The truth behind the Pentium Bug,'' Byte.com, March 1995; http://www.byte.com/art/9503/sec13/art1.htm.
3. S. Narayanasamy, B. Carneal, and B. Calder, ''Patching Processor Design Errors,'' *Proc. Int'l Conf. Computer Design* (ICCD 06), IEEE CS Press, 2006, pp. 491-498.
4. S. Sarangi, A. Tiwari, and J. Torrellas, ''Phoenix: Detecting and Recovering from Permanent Processor Design Bugs with Programmable Hardware,'' *Proc. Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 06), IEEE CS Press, 2006, pp. 26-37.
5. M.L. Corliss, E.C. Lewis, and A. Roth, ''DISE: A Programmable Macro Engine for Customizing Applications,'' *Proc. Ann. Int'l Symp. Computer Architecture* (ISCA 03), IEEE CS Press, 2003, pp. 362-373.
6. L.C. Heller and M.S.. Farrell, ''Millicode in an IBM zSeries Processor,'' *IBM J. Research and Development*, vol. 48, no. 3/4, May/July 2004, p. 425.
7. T.M. Austin, ''DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design,'' *Proc. Ann. ACM/IEEE Int'l Symp. Microarchitecture* (Micro 99), IEEE CS Press, 1999, pp. 196-207.
8. I. Wagner, V. Bertacco, and T. Austin, ''Shielding against Design Flaws with Field Repairable Control Logic,'' *Proc. Design Automation Conf* (DAC 06), ACM Press, 2006.
9. D.J. Sorin et al., ''SafetyNet: Improving the Availability of Shared-Memory Multiprocessors with Global Checkpoint/Recovery,'' *Proc. Ann. Int'l Symp. Computer Architecture* (ISCA 02), IEEE CS Press, 2002, pp. 123-134.
10. M. Prvulovic, Z. Zhang, and J. Torrellas, ''ReVive: Cost Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors,'' *Proc. Ann. Int'l Symp. Computer Architecture* (ISCA 02), IEEE CS Press, 2002, pp. 111-122.
11. J. Nakano et al., ''ReViveI/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers,'' *Proc. Int'l Symp. High-Performance Computer Architecture* (HPCA 06), IEEE CS Press, 2006, pp. 200-211.
12. S.P. Khatri, R.K. Brayton, and A. Sangiovanni, ''Cross-Talk Immune VLSI Design Using a Network of PLAs Embedded in a Regular Layout Fabric,'' *Proc. Int'l Conf. Computer Aided Design* (ICCAD 00), IEEE CS Press, 2000, pp. 412-418.

Smruti Sarangi is a PhD student in computer science at the University of Illinois, Urbana-Champaign. His research

interests include processor reliability, schemes to mitigate the effects of process variation, and power management schemes. Sarangi has a BTech in computer science and engineering from the Indian Institute of Technology, Kharagpur, and an MS in computer science from the University of Illinois. He is a student member of the IEEE.

Satish Narayanasamy is a PhD candidate in the Computer Science Department at the University of California, San Diego. His research interests include computer architecture support for system software and programming languages, programmer productivity tools, and fault tolerance. Narayanasamy has an MS from the University of California, San Diego, and a BE from Anna University, College of Engineering, Guindy, Chennai, India, both in computer science. He is a member of the ACM and the IEEE.

Bruce Carneal is a PhD student in the Computer Science Department at the University of California, San Diego. His research interests include computer architecture and machine learning. He received a BS in computer science from Brigham Young University.

Abhishek Tiwari is a PhD student in computer science at the University of Illinois, Urbana-Champaign. His research focuses on processor reliability and power management for deep submicron technologies. Tiwari has a BTech in computer science and engineering from the Indian Institute of Technology, Kanpur, and an MS in computer science from the University of Illinois. He is a student member of the IEEE.

Brad Calder is a professor of computer science and engineering at the University of California, San Diego. He has cofounded three startups (TracePoint, Entropia, and BitRaker), and is now an architect at Microsoft. He is co-editor-in-chief of *ACM Transactions on Computer Architecture and Code Optimization.* His research interests span distributed computing, architecture, compilers, and systems. Calder received his PhD in computer science from the University of Colorado, Boulder. He is an IEEE senior member and a member of the ACM.

Josep Torrellas is a professor of computer science and Willett Faculty Scholar at the University of Illinois, Urbana-Champaign. He is also chair of the IEEE Technical Committee on Computer Architecture (TCCA). His research interests include multiprocessor computer architecture, thread-level speculation, low-power design, and hardware and software reliability. Torrellas has a PhD in electrical engineering from Stanford University. He is an IEEE Fellow and a member of the ACM.

Direct questions and comments about this article to Brad Calder, University of California, San Diego, Dept. of Computer Science and Engineering, 9500 Gilman Dr., La Jolla, CA 92093-0404 (calder@cs.ucsd.edu); and to Josep Torrellas, Dept. of Computer Science, University of Illinois, 201 North Goodwin Ave., Urbana, IL, 61801 (torrellas@cs.uiuc.edu).