Shielding Against Design Flaws with Field Repairable Control Logic

Ilya Wagner Valeria Bertacco Todd Austin
Advanced Computer Architecture Lab, The University of Michigan – Ann Arbor, MI
{iwagner,valeria,austin}@umich.edu

ABSTRACT

Correctness is a paramount attribute of any microprocessor design; however, without novel technologies to tame the increasing complexity of design verification, the amount of bugs that escape into silicon will only grow in the future. In this paper, we propose a novel hardware patching mechanism that can detect design errors which escaped the verification process, and can correct them directly in the field. We accomplish this goal through a simple field-programmable *state matcher*, which can identify erroneous configurations in the processor's control state and switch the processor into formally-verified *degraded performance mode*, once a "match" occurs. When the instructions exposing the design flaw are committed, the processor is switched back to normal mode. We show that our approach can detect and correct infrequently-occurring errors with almost no performance impact and has approximately 2% area overhead.

Categories and Subject Descriptors. B.8.1 [**Performance and Reliability**]: Reliability and Fault-Tolerance; B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids-*Verification*
General Terms: Reliability, Verification
Keywords: Hardware patching, Processor verification.

1. INTRODUCTION

End-users of microprocessor-based products rely on the hardware to function correctly at all times. To meet this expectation, microprocessor design houses perform extensive validation of their designs before production and release to the marketplace. The success of this process is crucial to the survival of the company, as the financial impact of microprocessor bugs can be devastating (e.g., the infamous Pentium FDIV bug, which cost Intel $475 million).

Designers address correctness concerns through verification, the process of extensively validating all the functionalities of a circuit throughout the development process. Simulation-based techniques are central to this process: they exercise a design with relevant test sequences trying to expose latent bugs. However, this approach is often incapable of fully exercising the design space of modern processors. For example, the simple out-of-order core that we use in the experiments for this work has a total of 2^{10441} distinct states, each with up to 2^{128} outgoing edges. In contrast, the verification of the Pentium 4, which used a pool of 6,000 workstations, was only able to test as many as 2^{37} states prior to tape-out [4]. It is obvious, that engineers must be very selective in the configurations that they choose to validate.

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

Formal verification techniques have grown to address the non-exhaustive nature of simulation-based methods. Formal methods utilize mechanisms such as theorem proving and model checking to show that a component violates or upholds a certain property. The primary drawback of formal techniques, however, is that they do not scale to the complexity of modern designs, constraining their use to only a few components within the processor. For example, the verification of the Pentium 4 heavily utilized formal verification techniques, but their use was limited to proving properties of the floating-point execution units, the instruction decoders, and the dynamic scheduler [5].

Unfortunately, the situation is deteriorating with exponentially increasing design complexity and slower growth rate in the capabilities of design verification tools. Thus, without better verification solutions, or techniques to shield the system from design errors, we can only expect future designs to be more and more flawed.

In this paper, we introduce an expressive, reliable and low-cost control logic patching mechanism that allows a wide range of control-logic design bugs in a processor pipeline to be fixed in the field after manufacturing.

Our approach employs a control state matching mechanism, the *state matcher*, that identifies when the processor has entered a control state associated with a design bug. Once a match occurs, the pipeline is flushed and forced into a *degraded performance mode* of operation to execute the next instruction. We formally verified the correctness of the system in this mode of execution, guaranteeing correct forward progress past the occurrence of the flawed configuration by running in degraded mode. We specifically designed the control state matcher to detect multiple design errors with minimal false-positive triggering.

In the experimental result section we show that the area and performance impact of our solution are minimal. When multiple design errors must be corrected, we show that our state matcher can effectively trade-off area for false positive rate. Our solution goes beyond instruction and microcode patching because it can effectively address design errors that relate to a particular instruction, combination of instructions, and even errors that are not associated with any specific instruction, for instance a non-maskable interrupt.

The remainder of the paper is organized as follows. Section 2 makes a case for repairable control logic, by examining types of bugs that escape verification. Section 3 details our approach to field-repairable control logic and Section 4 evaluates its performance and area overhead. Finally, we present our conclusions in Section 5.

2. ESCAPED BUGS AND IN-FIELD REPAIR

In this section we present an analysis of design errors in modern commercial microprocessors and the main approaches used to fix them in the field.

2.1 Escaped errors in commercial processors

We studied a number of escaped errors reported for ARM[2], x86[1, 9], and PowerPC[3] processors. We show here that a large fraction of them is related to the control portion of the design. The results of the study are summarized in Table 1. Errors are classified into one of the following categories:

Processor's Control Logic: These are the escape bugs addressed by this work, they are the result of incorrect decisions made at the occurrence of important execution events and bad interactions between simultaneous events. For example, in the early 486 processors, two simultaneous events of writing to register TR5 and a pending memory prefetch would cause the processor to hang [9].

Functional Units: These are errors in the design of a functional unit which cause it to produce an incorrect result, including bugs in components such as branch predictors and TLBs. An (infamous) example of this type of error is the flawed lookup table in Pentium FDIV bug [1].

Memory System: These are bugs in the the on-chip memory systems, caches, and memory interfaces.

Microcode: These are (software) bugs in the implementation of the microcode for a particular instruction.

Electrical faults: These are design errors occurring when certain logic paths do not meet timing under exceptional conditions. Consequently, if a processor is run well below the specified maximum frequency, these faults will often not occur. An example is the Load Register Signed Byte (LDRSB) instruction of the StrongARM SA-1100 which does not meet timing when reading from the pre-fetch buffer [2].

As we discussed above, control logic escapes dominate the errata reports of the processors. The high frequency of such escapes is due to the complexity of the processor's control logic. Related studies on the sources of design errors corroborate our finding. For example, Van Campenhout's work [6] reported that many flaws were the result of incorrectly implemented interactions between major modules.

Bug type	Occurrences	Incidence
Processor's control logic	19	52%
Functional units	3	8%
Memory system	7	19%
Microcode	2	5%
Electrical faults	6	16%
Total	37	100%

Table 1: Classification of escaped design errors reported in [1,2,3,9]

2.2 Related in-field repair solutions

Currently the two main approaches for correcting design flaws in the field are instruction and microcode patching.

Instruction Patching: Software patching can sometimes correct the execution of an erroneous instruction [11]. In this approach, the program code is inspected and, if a broken instruction is encountered, it is replaced with an alternative implementation, typically a procedure that emulates the instruction. This technique was used as the initial workaround for the Pentium FDIV bug [11].

Microcode Patching: Intel and AMD processors reportedly have the ability to update their microcode after tapeout [10, 8, 7]. During the system startup, microcode patches are loaded into an on-chip buffer, which overrides existing microcode in on-chip ROMs. A microcode patch can change the semantics of any instruction, similar to instruction patching; however, no changes to the binary are needed, since the patching occurs in hardware in decode stage.

While these techniques have proven their positive impact in commercial solutions, their value is impaired by their inability to cope with complex control bugs and by the potential cost in performance. For example, with the Pentium FDIV, all divide instructions had to be replaced with an emulated version, resulting in significant slowdowns. Additionally, many control logic bugs cannot be easily bound to a particular instruction, and thus they could not be fixed by any of these techniques. For instance, in the 486 processor, if a non-maskable interrupt (NMI) occurred at the same cycle as a global segment violation, the violation would not be detected [1]. Short of emulating every instruction, this bug could not be fixed with instruction patches.

3. FIELD REPAIRABLE CONTROL LOGIC

Figure 1 illustrates our approach to correcting design errors in the field. We complement the processor with a field-programmable state matcher, which by default is empty. If an error is found after design tape-out, the set of states associated with the bug are encoded into a state matching pattern. This pattern is then distributed to customers, where it is loaded into the state matcher. When a buggy state is detected by the state matcher, the processor switches to a degraded mode with formally verified execution semantics. Thus, we can rely on this mode for the processor to complete the next instruction correctly, guaranteeing forward progress. After the instruction is committed, normal execution mode is resumed.

3.1 Matching flawed configurations

The correct state transition graph (STG) of a device consists of all the legal states (configurations of internal elements) S_i. The states are connected by the legal transitions between them. In this framework an error is represented by an erroneous transition from a legal state to an illegal state, or an invalid transition between two legal states, or by the lack of a transition that should exist. In our field-repairable control logic solution, we add hardware support which allows us to detect *erroneous states*, that is, states which are sources of illegal transitions.

Our state matcher is implemented as a content-addressable memory (CAM), loaded with a control state pattern, which is matched against the current pipeline state at every cycle. The state matcher can be thought of as a fully-associative cache with the width of the tag being equal to the entire control state vector. For each control configuration, if such a tag exists in the cache, then a hit occurs. We also modified the CAM to allow for *don't care* bits in the state vector to be matched. Essentially these don't care bits mask out some of the bits in the state when they are compared to the specified bits in the CAM entry. In other words, the string representing an entry in the CAM can be specified in a format similar to the following: $011xxxx11xx0x1$. In this case 0's and 1's represent the fixed value bits in the state, while x's represent the don't cares in the entry and they can match any value in the corresponding control state bit.

Constructing the state patterns for the matcher is a two-step process. First, all relevant erroneous states are identified when a new bug is found. A specific pattern is then constructed for each of these states: all relevant control bits

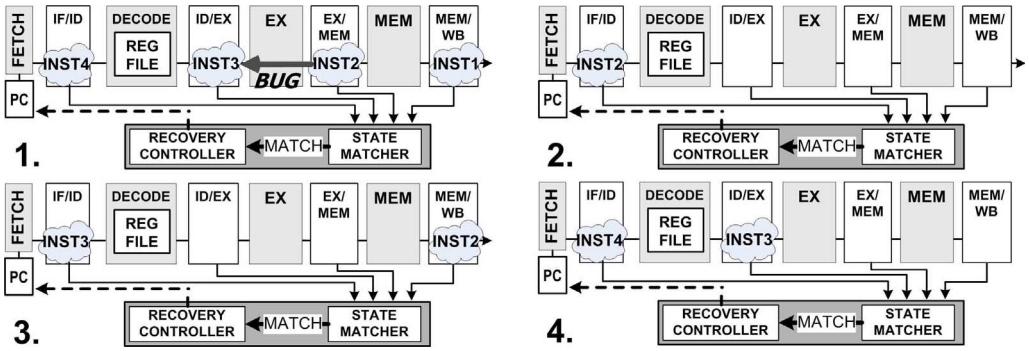

Figure 1: Field Repairable Control Logic. 1. Matcher detects a state associated with a bug. 2. Pipeline is flushed to a known state. 3. Processor runs in degraded mode. 4. Processor resumes normal mode operation.

are specified, while the remaining are left as don't cares. If this step generates fewer patterns than the number of entries in the matcher, then the patterns are simply loaded into it.

However, if the number of patterns to be uploaded to the matcher is larger than the the number of entries available, then we need to apply a compression mechanism. This situation may arise when a bug affects a large number of configurations or when multiple bugs need to be corrected.

The state compression algorithm maps a number k of state patterns into a r-entries state matcher CAM. To do so, it first builds a *proximity graph* where each vertex represents a pattern, and edges connecting vertices are assigned a variant of the hamming distance metric. Specifically, we compare the corresponding bits of the patterns and each $0-1$ pair contributes 1 to the weight, while each $1-x$ or $0-x$ pair contributes 0.5 to the weight. As an example, the two patterns $101xx1$ and $1001x1$ would have a connecting edge with weight 1.5. The reasoning behind this weighing system is fairly straightforward: if the two patterns above were to be compacted into a single entry, it would have to be encoded as $10xxx1$. So, each discording bit pair contributes the same amount of approximation in the entry generated. However, pairs such as $1-x$ or $0-x$, have an approximating impact only on one of the patterns (the one with the 0 or 1).

Once the proximity graph is built, the two patterns connected by the minimum-weight edge are merged together. If $r = k - 1$, the compression is completed, otherwise a new proximity graph is built and the process is repeated until the number of patterns is small enough to fit in the CAM.

Note also that the compression algorithm generates patterns that *over-approximate* the number of erroneous configurations. The resulting state matcher will still be capable of flagging all the erroneous configurations, *i.e.* it never produces *false negatives*, however, it will also flag additional (*false positive*) configurations that arise due to merging. The impact on the overall system will not be one of correctness, but one of performance. We measure the amount of approximation in the matcher as its *specificity*. The specificity is the probability that a state matcher will not flag a correct control state configuration as erroneous. Specificity can also be thought of as $1 - false\_positive\_rate$.

3.2 Processor recovery

Once a bug state has been identified, the processor is forced into a formally verified degraded performance mode, which executes one instruction and then resumes the normal mode. By finishing one instruction reliably before returning to normal operation, forward progress in the presence of any patched bug is guaranteed. To strengthen the recovery mechanism, we formally verified the degraded mode of operation of our design. Since only one instruction will be present in the pipeline at a time in recovery mode, the control logic is greatly simplified, making formal verification possible. In our experiments we used Magellan from Synopsys to verify the degraded mode of operation of the processor designs used in our experiments.

4. EXPERIMENTAL EVALUATION

In this section we detail a prototype system with field-repairable control logic support. Using simulation-based analysis, we examine the specificity of our design for a number of design error scenarios and varied state matcher storage sizes. In addition, we examine the area costs of adding this support to a simple microprocessor. Finally, we examine the performance impact of degraded mode of execution, to evaluate the extent of error recovery which can be tolerated before the performance degradation becomes apparent.

4.1 Experimental Framework

To gauge benefits and costs of our field-repairable control logic, we inserted it in two prototype processors that we used in earlier research projects. The first processor design (In-Order) is a 5-stage in-order pipeline implementing a subset of the Alpha ISA. For this design, the state vector passed to the matcher consisted of 26 control bits, including logic governing forwarding, branch misprediction, memory operations, and ALU functions.

The second processor (Out-of-Order) is a larger out-of-order 2-way super-scalar pipeline also implementing a subset of the Alpha ISA. The design has four reservation stations for each of the function units and 32 re-order buffer (ROB) entries to hold speculative results. The flushing of the core on a branch mispredict is performed when the branch reaches the head of the ROB. The state vector sent to the state matcher consists of signals from the retirement logic in the ROB, as well as control signals from reservation stations and renaming logic. Both cores were outfitted with 256 byte direct-mapped instruction and data caches and a global-history branch predictor. For performance analysis, we ran a battery of programs, designed to fully exercise the processor while providing small code footprints. For both designs we used state matchers with 4 and 8 entries.

4.2 Design Defects

To test the performance of our field-repairable control logic, we manually inserted a variety of bugs into our de-

signs, and then examined the performance of the designs operating with field-repaired control logic. The high-level bugs consisted of bad interactions between multiple instructions in the pipeline. For example, *opA-forward-wb* breaks forwarding from WB stage on one operand, and *2-branch-ops* prevents two consecutive branching operations from being processed properly. Medium-level bugs introduced incorrect handling of instruction computations, such as *store-mem-op*, which causes store operations to fail. Low-level bugs where highly-specific failure scenarios, for example, *r31-forward* is a bug in which forwarding on register 31 is performed incorrectly. Finally, the multi-bugs are examples of combined bugs, where the state matcher is required to recognize multiple bug states. For example, *multi-all* is a design that contains all bugs simultaneously.

4.3 Matcher Specificity Under Varied Load

Figure 2 graphs the specificity of the state matcher for bugs in the In-Order and Out-of-Order processor designs. Recall that the specificity is the fraction of recoveries that are due to an actual bug. Thus, if the specificity is 1 the state matcher only recovers the machine when the bug is encountered. And if the specificity is 0.80, then 20% of the time the machine is recovered, the bug did not occur.

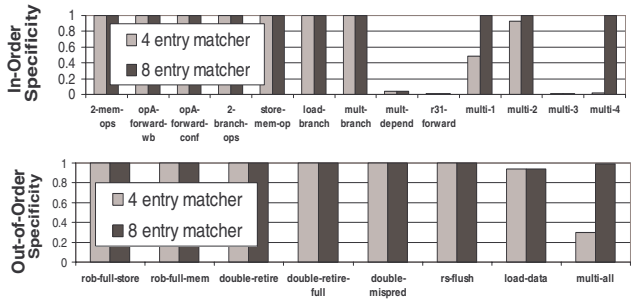

Figure 2: Specificity of Design Error Detection

It can be noted that, for many of the bugs, the specificity of either matcher design is 1.0, thus no spurious recoveries were initiated. Also, some combinations of multiple bugs (*e.g.*, *multi-1* and *multi-2*) had low specificities until the size of the state matcher was increased. For these combinations of bugs, a four entry CAM was too small to accurately describe the state space associated with them. Finally, for some of the bugs, *e.g.*, *mult-depend* and *load-data*, even a larger CAM did not improve specificity. This however was not caused by the pressure on the CAM, but rather insufficient access to critical control state. Consequently, these experiments had to initiate recovery whenever a potential error would occur, which led to the lower specificities.

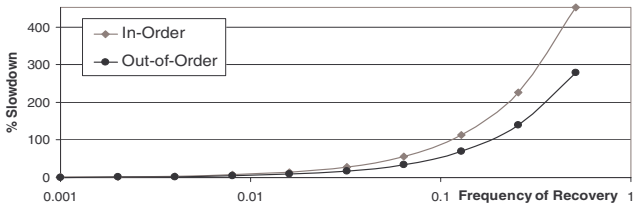

Figure 3: Performance Impact of Recovery

4.4 State Matcher Area Overheads

Field-repairable control logic requires the addition of the control state matcher, resulting in some area overhead. To calculate it, both the In-Order and Out-of-Order designs were synthesized and mapped to Artisan standard cell logic in a TSMC 0.18um technology. We found that the 4-entry matcher incurred a 0.02% overhead for Out-of-Order design with 64kB cache and 1.10% overhead for In-Order design with 256B cache. The 8-entry matcher incurred 0.02% and 2.20% area overhead respectively. Given the simplicity of our designs, we would expect the overheads for commercial designs to be even lower.

4.5 Performance Impact of Recovery

In the event the state matcher identifies a bug state, the processor is switched into recovery mode for one instruction, after which the pipeline is returned to normal operation. During recovery, only one instruction at a time is permitted in the pipeline, thus instruction-level parallelism is lost and program performance suffers accordingly. Figure 3 graphs the performance of the In-Order and Out-of-Order processors as a function of increasing recovery frequency. As shown in the graph, program performance is not adversely impacted until the rate of recovery is 1 out of 100 cycles, after which the performance impact rises quickly. In addition, the In-Order design is affected sooner by recovery than the Out-of-Order processor, due to the fact that the Out-of-Order core is able to better tolerate the loss of parallelism with its more capable instruction scheduler.

5. CONCLUSIONS

In this paper we introduced the concept of field-repairable control logic, and presented a design that can detect when a processor enters a bug state and switch to a low-complexity reliable execution mode until the bug is bypassed. We described a low-cost CAM-based state matching mechanism to detect and recover from bugs. With moderate size matchers we can ensure highly accurate detection of bug states, as nearly all of our experiments demonstrate a specificity of 1. We found the area cost of the technique to be about or below 2%. Finally, we found that if the bug matching frequency is less than one recovery per 100 instructions, performance impacts are negligible.

6. REFERENCES

[1] DDJ Microprocessor Center. `http://www.x86.org/`.

[2] *Intel(R) StrongARM(R) SA-1100 Microprocessor Specification Update*, Feb. 2000.

[3] *IBM PowerPC 750GX and 750GL RISC Microprocessor Errata Notice*, July 2005.

[4] B. Bentley. Validating a modern microprocessor. In *Proc. CAV*, July 2005.

[5] B. Bentley and R. Gray. Validating the Intel Pentium 4 Microprocessor. *Intel Technology Journal*, pages 1–8, 2001.

[6] D. V. Campenhout, T. Mudge, and J. P. Hayes. Collection and analysis of microprocessor design errors. *IEEE Design & Test*, 17(4):51–60, 2000.

[7] A. Carbine. *U.S. Patent no. 5253255: Scan mechanism for monitoring the state of internal signals of a VLSI microprocessor chip*, Nov. 1990.

[8] J. K. P. Kevin J. McGrath. *U.S. Patent no. 6438664: Microcode patch device and method for patching microcode using match registers and patch routines*, Oct. 1999.

[9] D. Koncaliev. Bugs in the Intel Microprocessors. `http://www.cs.earlham.edu/~dusko/cs63/`.

[10] D. S. C. Michael D. Goddard. *U.S. Patent no. 5796974: Microcode patching apparatus and method*, Nov. 1995.

[11] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. *SIGPLAN Not.*, 39(4):528–539, Apr. 2004.