Using Field-Repairable Control Logic to Correct Design Errors in Microprocessors

Ilya Wagner, *Student Member, IEEE*, Valeria Bertacco, *Member, IEEE*, and Todd Austin, *Member, IEEE*

***Abstract*—Functional correctness is a vital attribute of any hardware design. Unfortunately, due to extremely complex architectures, widespread components, such as microprocessors, are often released with latent bugs. The inability of modern verification tools to handle the fast growth of design complexity exacerbates the problem even further. In this paper, we propose a novel hardware-patching mechanism, called the field-repairable control logic (FRCL), that is designed for in-the-field correction of errors in the design's control logic—the most common type of defects, as our analysis demonstrates. Our solution introduces an additional component in the processor's hardware, a state matcher, that can be programmed to identify erroneous configurations using signals in the critical control state of the processor. Once a flawed configuration is "matched," the processor switches into a degraded mode, a mode of operation which excludes most features of the system and is simple enough to be formally verified, yet still capable to execute the full instruction-set architecture at one instruction at a time. Once the program segment exposing the design flaw has been executed in a degraded mode, we can switch the processor back to its full-performance mode. In this paper, we analyze a range of approaches to selecting signals comprising the processor's critical control state and evaluate their effectiveness in representing a variety of design errors. We also introduce a new metric (average specificity per signal) that encodes the bug-detection capability and amount of control state of a particular critical signal set. We demonstrate that the FRCL can support the detection and correction of multiple design errors with a performance impact of less than 5% as long as the incidence of the flawed configurations is below 1% of dynamic instructions. In addition, the area impact of our solution is less than 2% for the two microprocessor designs that we investigated in our experiments.**

***Index Terms*—Hardware patching, processor verification.**

I. Introduction

End-users of microprocessor-based products rely on the hardware system to function correctly all the time for every task. 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 resulted in a $475-million cost to Intel to replace the defective parts).

Designers address correctness concerns through verification, which is the process of extensively validating all the functionalities of a design throughout the development cycle. Simulation-based techniques are central to this process: They exercise a design with relevant test sequences in an attempt to expose latent bugs. This approach is used extensively in the industry, yet it suffers from a number of drawbacks. First, a simulation-based verification is a nonexhaustive process: The density of states in modern microprocessors is too large to allow for the entire state space to be fully exercised. For example, the simple out-of-order processor core that we use as our experimental platform throughout this paper has 128 input signals, 31 64-b registers, and additional control states for a total of 2^{10441} distinct configurations, each with up to 2^{128} outgoing edges connecting to other configurations. In contrast, the verification of the Pentium 4, which utilized a simulation pool of 6000 workstations, was only able to test 2^{37} states prior to tape-out [12]. It is obvious from this disparity that verification engineers must be extremely selective in the set of configurations that they choose to validate before tape-out.

Formal verification techniques have grown to address the nonexhaustive nature of simulation-based methods. Formal methods (such as theorem provers and model checkers) enable an engineer to reason about the correctness of a hardware component, regardless of the programs and storage state impressed upon the design. In the best scenario, it is possible to prove that a design will not exhibit a certain failure property or that it will never produce a result that differs from a known-correct reference model. 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 overall design. For example, the verification of the Pentium 4 heavily utilized formal verification tools, but their use was limited to proving properties of the floating-point units, the instruction decoders, and the dynamic scheduler [13].

Unfortunately, the situation seems to be deteriorating in the presence of seemingly unending design complexity scaling, in contrast with a much slower growth of the capabilities of verification tools, leading to what is referred to as the "verification gap" [11]. In the end, processor designs are released not fully tested and, hence, with latent bugs, as we show in Section II-A. In addition, 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.

A. Contributions of This Paper

In this paper, we introduce a reliable, low-cost, and extremely expressive control-logic-patching mechanism for microprocessor pipelines, which enables the correction of a wide range of

Manuscript received March 7, 2007; revised May 23, 2007. This paper was recommended by Associate Editor W. Kunz.

The authors are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122 USA (e-mail: iwagner@umich.edu).

Digital Object Identifier 10.1109/TCAD.2007.907239

control-logic-related design bugs in parts deployed in the field after manufacturing. In our framework, when an escaped bug is found in the field, the support team investigates it and generates a pattern describing the control state of the processor which causes the bug to manifest itself. The pattern is then sent to the end customers as a patch and is loaded into the on-die state matcher at startup. The matcher constantly monitors the state of the processor and compares it to the stored patterns to identify when the pipeline has entered a state associated with a bug. Once the matcher has determined that the processor is in a flawed control state, the processor's pipeline is flushed and forced into a degraded mode of operation.

In the degraded mode, the processor starts the execution from the first uncommitted instruction and allows only one operation to traverse the pipeline at a time. Therefore, much of the control logic that handles interactions between operations can be turned off, which enables a complete formal verification of the degraded mode at the design time. In other words, we can guarantee that instructions running in this mode complete properly and, thus, can ensure forward progress, even in the presence of design errors by simply forcing the pipeline to run in the degraded mode. After the error is bypassed in the degraded mode, the processor returns to a high-performance mode until the matcher finds another flawed control state. In designing the state matcher, we have put special care into creating a system that can detect multiple design errors with minimal false-positive triggering. In addition, for cases when the number of patterns of design errors exceeds the capacity of a given matcher, we developed a novel pattern-compression algorithm that compacts the erroneous state patterns while minimizing the number of false positives introduced by this process. Our solution makes strides past the capabilities of instruction and microcode patching because it can effectively address errors that relate to a single instruction or combination of instructions, and even errors that are not associated with specific instructions, for instance a nonmaskable interrupt (NMI).

A preliminary version of this paper was published in [27]. In this paper, we substantially extend our analysis of the proposed approach, including a detailed performance evaluation of a range of solutions with matchers observing distinct sets of control signals. In addition, we investigate a metric to compare different solutions based on their effectiveness in recognizing a variety of design errors and the number of monitored signals. We also present a novel algorithm to automatically select control signals that operates directly on a register-transfer-level (RTL) design description. Finally, this paper presents a novel pattern-compression algorithm and a detailed explanation of how the degraded mode is formally verified.

The remainder of this paper is organized as follows. Section II makes the case for new technology which allows to repair control-logic faults in a design after shipment and deployment by examining the type of bugs that escape verification. Section III details the flow of operation of the proposed approach, whereas Section IV presents the general framework in which a repairable logic can be used. Section V details the experimental setup and evaluates the performance of the matching mechanisms, including the accuracy and performance impacts. Finally, Section VI concludes this paper.

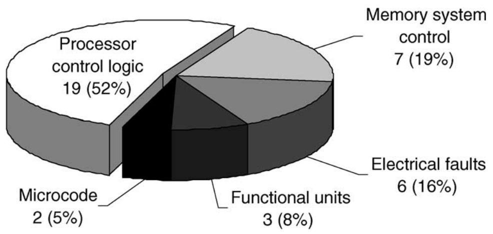

Fig. 1. Classification of escaped bugs found in x86 [1], [4], [5], [7], [9], [10], [22], StrongARM-SA1100 [3], and PowerPC 750GX [8] processors. The chart shows occurrences and incidence of each particular type of bug.

II. Escaped Bugs and In-Field Repair

A. Escaped Errors in Commercial Processors

Despite the impressive efforts of processor design houses to build correct designs, bugs do escape the verification process. In this section, we examine the reported escaped errors of a number of commercial processors. We classify these bugs and show that a large fraction of them are related to the control portion of the design. The summary of the bugs reported in x86 [1], [4], [5], [7], [9], [10], [22], StrongARM-SA1100 [3], and PowerPC 750GX [8] processors is shown in Fig. 1. Errors are classified into one of the following categories.

Processor's control logic: These bugs are the result of incorrect decisions made at the occurrence of important execution events and also of bad interactions between simultaneous events. An example of this type of escape could be found in the Opteron processor, where a reverse REP MOVS instruction may cause the following instruction to be skipped [10]. Our solution addresses precisely these types of bugs.

Functional units: These are design errors in units which can cause the production of an incorrect result. In this category, we included bugs in components, such as branch predictors and translation lookaside buffers. An (infamous) example of this type of bug is the Pentium FDIV bug, where a lookup table that is used to implement a divider Sweeney, Robertson, and Tocher algorithm contained incorrect entries [1].

Memory system control: These are bugs that occur in the on-chip memory system, including caches, memory interface, etc. An example of this type of bug is an error in the Pentium III processor, where certain interactions of instruction fetch unit and data cache unit could hang the system [9].

Microcode: These are (software) bugs in the implementation of the microcode for a particular instruction. An example can be found in the 386 processor, where the microcode incorrectly checked the minimum size of the task state segment, which must be 103 B, but due to a flaw, segments of 101 and 102 B were also incorrectly allowed [22].

Electrical faults: These are design errors occurring when certain logic paths do not meet timing under exceptional conditions. Consequently, if a processor runs well below its specified maximum frequency, these faults will often not occur. An example is the load register signed byte instruction of the StrongARM SA-1100 which does not meet timing when reading from the prefetch buffer [3].

As the aforementioned analysis demonstrates, control-logic escapes dominate the errata reports for these processors. The high frequency of such escapes can be explained by the complexity of the control-logic blocks that handle the interactions between multiple instructions and the inability of formal techniques to handle complex interactions between multiple logic blocks in a design. Correctness of the datapath, on the other hand, can frequently be proven formally. In our framework, we utilize this capability to prove the datapath's correctness when no control-logic interactions are present in the system. Related studies on sources of the design errors corroborate our findings, an example being the work of Van Campenhout *et al.* [15], reporting that many design flaws are the result of incorrect interactions between major components or an unforeseen combination of rare events.

B. Related In-Field Repair Solutions

Given the high incidence of the escaped design errors and also their associated risk of causing a very negative impact on the survival of a company, over the past few years, processor manufacturers have started to explore solutions that could correct a design error in the field. To date, we are aware of two techniques in this domain, which have been deployed commercially.

Instruction Patching: Software patching can sometimes correct the execution of an instruction which has an erroneous implementation [25]. In this approach, the program code is inspected, and if a broken instruction is encountered, it is replaced with an alternative implementation, typically through a function call to a correct emulation of the instruction. Consequently, each occurrence of the instruction will be emulated.

This technique was used as the initial workaround for the Pentium FDIV bug using software recompilation. Linux- and Windows-based compilers were updated to generate code which would run a preliminary test to determine if the underlying processor suffered from the FDIV bug. If the test indicated so, a divide emulation routine would be called to avoid the use of a hardware divider [25]. A similar technique was used to port Windows NT to Alpha processors [14]: A bug in the underflow exception mechanism forced Alpha software developers to make the operating system step in and handle the offending instructions in the software. A specific advantage of this approach was that it could operate in a completely transparent fashion to the user (besides the requirement of installing an operating system patch). Performance wise, however, this approach is not very promising. For example, the FDIV fix [2] in the Microsoft Visual C++ compiler incurs 100% worst-case performance overhead on a flawed processor. Moreover, on a correctly working chip, it still causes up to 10% overhead.

Microcode Patching: Intel and AMD processors reportedly have the ability to update their microcode after deployment in the field [16], [21], [23].[1] During system startup, microcode patches are loaded into a small on-chip buffer, which overrides the existing microcode in on-chip ROMs. A microcode patch can change the semantics of any instruction, which is similar to the instruction patching. An added advantage of the microcode patching is that no changes are necessary to the existing software since patching occurs during the instruction's decode stage. The concept of patchable microcode is not new, as many early computers such as the Xerox Alto and DEC LSI-11 supported writable microstores, thus allowing engineers to update the implementation of individual instructions [18].

While these techniques have proven their positive impact in commercial solutions, they have a limited value because of their high performance impact and due to their inability to cope with complex control bugs. For example, in the case of the Pentium FDIV bug, all divide instructions had to be tested for susceptibility to the bug and replaced with an emulated routine if needed, which resulted in significant slowdowns. In addition, many control bugs are not associated with a particular instruction, and thus, they could not be fixed with any of these techniques. For example, on the 486 processor, if a non-NMI occurred in 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.

A related work by Sarangi *et al.* [24], which appeared after the initial publication of our solution in [27], suggests a similar mechanism for hardware patching. An error in this work is identified by its fingerprint: a set of conditions and a time interval during which these conditions are satisfied when the error occurs. Similar to our work, this mechanism relies on internal signals being observed by the programmable error-checking module. However, the matcher in [24] is distributed and contains multiple modules that detect the occurrence of various events and identify if they correspond to an error. The work also proposes several recovery mechanisms, including dynamic microcode editing, checkpointing, and hypervisor support. Unfortunately, it is unclear how much performance overhead these techniques would have since they require either the complex hardware for microcode editing and checkpointing or the inclusion of trapping to software hypervisor. Another technique for recovery mentioned in [24] is similar to our work and requires flushing the pipeline and replaying the instruction stream. However, unlike field-repairable control logic, the replay is not done in a reliable mode; hence, it does not guarantee that the bug will be bypassed. Finally, the patching technique in [24] potentially incurs a higher area overhead due to the distributed nature of the detection blocks but may allow for better recovery from bugs exposed by long-event sequences.

III. Flow of Operation

This section presents the usage flow and the process to correct escaped bugs for a design incorporating the FRCL technology. We also show the structure of the state-matcher circuit and present a pattern-compression algorithm for cases when the number of patterns exceeds the size of the matcher. Finally, we analyze an example of an actual bug that is repaired using our approach.

[1]In fact, neither company will disclose the details (or even the existence) of microcode-patching infrastructure due to concerns that they could be exploited by virus writers. However, evidence of the infrastructure is well hinted to by the patent literature.

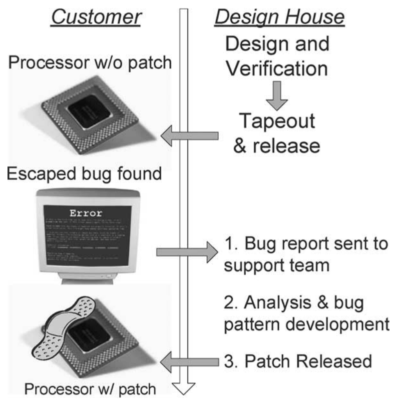

Fig. 2. FRCL usage flow: After a component is shipped to the end customer and a new bug is found, a report detailing the bug is sent to the support team. The error is analyzed, and patterns representing the control states associated with the bug are issued as a patch. On every startup, the processor loads the patterns into the state matcher, and if a bug is encountered, it is bypassed through the reliable degraded mode.

The FRCL is designed to handle flaws in processor control circuitry for components already deployed in the field. The flow of operation that we envision for this approach is shown in Fig. 2. When an escaped error is detected by the end customer, a report containing the error description, such as the sequence of executed operations and the values in the status registers, is sent to the design house. Engineers on the product support team investigate the issue, identify the root cause of the error and which products are affected by it, and decide on a mechanism to correct the bug. As previously mentioned, the instruction and microcode patchings are valid approaches; however, they can have a very high performance overhead or can be too costly. We propose that the engineers use instead our solution—the FRCL. By knowing the cause of the bug and which signals are monitored by the matcher in the defective processors, the engineers can create patterns that describe the flawed control-state. The patterns then can be compressed by the algorithm presented in Section III-C and can be sent to the customers as a patch. The patches in the end system are loaded into the state matcher at startup. Every time the patched error is encountered at runtime, a recovery via a degraded mode, which is detailed in Section III-D, is initiated, effectively fixing the bug.

A. Pattern Generation

The pattern to address a design error can be created from the state transition graph (STG) of a device. The correct STG consists of all the legal states of operation, where each state is a specific configuration of internal signals that are crucial to the proper operation of the device. In addition, these states are connected by all the legal transitions between them. Within this framework, an error may occur because of an additional erroneous transition from a legal state to an illegal state, which should not be part of the STG, or when an invalid transition connects two legal states, or by the lack of a transition that should exist between states (Fig. 3). In our solution, we add a hardware support that uses patterns to detect both the illegal states and the legal states which are sources of illegal transitions. A pattern is a bit vector representing the configuration of the internal signals, which is associated with an erroneous behavior of the processor. Note that, in this framework, a single bug can be mapped to multiple patterns if it is caused, for example, by multiple illegal states. To cope with this problem, we incorporated a range of features into our technology, including a novel pattern-compression algorithm presented in Section III-C. In a real-world scenario, after receiving a bug report, a product support team would analyze the issue, try to reproduce the error, and understand what caused it. Tools such as trace minimizers can be very helpful for this analysis since they can significantly shorten a trace that leads to a bug, which helps immensely in the debugging process. Moreover, some of these tools, for example Butramin [20], investigate alternative simulation scenarios that reach the same bug. This allows the support team to pinpoint multiple processor control states associated with the bug and to identify how these states map to the critical signals observed by the matcher in the design. Afterward, the configurations of the critical control signals are compactly encoded and issued as a patch to the end customer. The process is repeated when new bugs or new scenarios exposing the known bugs are discovered.

B. Matching Flawed Configurations

As mentioned before, the design errors and patterns describing the bugs in our framework are defined through the configurations of control signals of the processor and through the transitions between these configurations. At runtime, these signals are continuously observed by a state matcher and are compared to preloaded patterns describing the bugs. Therefore, only the bugs that manifest themselves on these critical signals can be detected by the matcher. Ideally, all of the design's control signals could be used for this purpose; however, complexity and stringent timing constraints of modern chips prevent such extensive monitoring, allowing only a small portion of the actual control state to be routed to the matcher. In Section IV-C, we present techniques to intelligently select these critical state bits among the prohibitively large control state of a processor.

The state matcher can be thought of as a fully associative cache, with the width of the tag being equal to the width of the critical control-state vector, which, in our experiments, was just several tens of bits long. The tag in this case is the pattern describing an erroneous configuration; thus, if such a tag exists in the cache, then a hit occurs and a potential bug is recognized. In order to improve the performance of the matcher, we structured it to allow the use of don't care bits in the patterns to be matched. The don't care bits help make a compact representation of multiple individual configurations of the critical control state, which differ in just a few bits. By using our state matcher, designers issuing a patch can specify a bug pattern through a vector of 0s, 1s, and don't care bits (x): 0s and 1s represent the fixed value bits, whereas x's can match any value in the corresponding control signal. Note, however, that the control state observed by the matcher at runtime contains only the fixed bit values 0 and 1. Fig. 8 shows several examples of bug patterns loaded into a four-entry matcher.

We also anticipate that a single patch may consist of multiple bug patterns since a single bug may be associated with

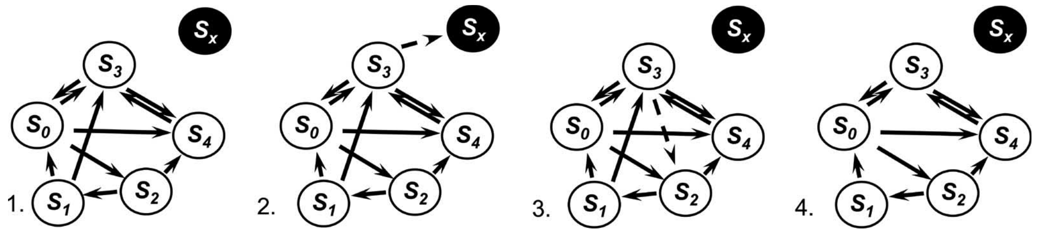

Fig. 3. Error representation in the STG framework. (1) Correct STG of the device (S_x is an unreachable illegal state). (2) Erroneous STG due to a transition to an illegal state S_x. (3) Erroneous STG due to an illegal transition between legal states S_3 and S_2. (4) Erroneous STG due to the absence of a legal transition $S_1 \rightarrow S_3$.

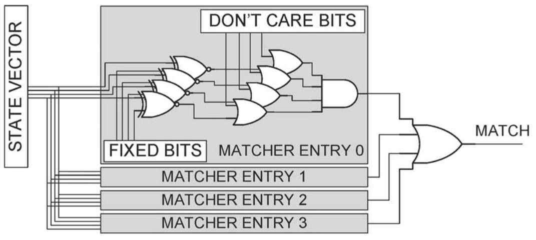

Fig. 4. State matcher. The critical control-state vector is first compared against the fixed bits in a bug pattern. Then, the don't care bits in the pattern are overlaid, and the result is reduced to a single match bit. The matcher contains multiple independent entries to allow for multiple simultaneous comparisons.

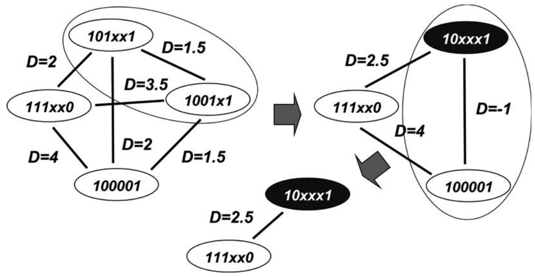

Fig. 5. Pattern-compression example: Four bug patterns are compressed to fit into a two-entry matcher. A complete graph of the initial four patterns is computed and is labeled with a variant of distance. The first compression step combines the two closest (in terms of distance) patterns $101xx1$ and $1001x1$. The resultant pattern $10xxx1$ has fixed bits in every position where original patterns were identical and have don't care bits (x) in all other positions. In the second step, pattern 100001 is eliminated, since it is a subset of the pattern $10xxx1$, as the -1 label indicates.

several patterns, as aforementioned, or the design may contain multiple unrelated bugs. To handle this situation, we developed a matcher with multiple independent entries, as shown in Fig. 4. On startup, each of the matcher's entries is loaded with an individual pattern containing fixed bits and don't cares. At runtime, the matcher simultaneously compares the actual critical control bit values to all of the valid entries and asserts a signal if at least one match occurs. The number of entries in the matcher is set at design time and is one of the engineering tradeoffs. A larger matcher can be loaded with more patterns; however, it has a larger area on the die and longer propagation delay. A smaller matcher, on the other hand, might not be able to load all of the patterns, and compression would be needed.

C. Pattern-Compression Algorithm

The pattern-compression algorithm that we developed was inspired by the two-level logic minimization techniques described in [17]. Our algorithm compresses a number k of patterns into a state matcher with r entries, where $k > r$. This process, however, often overapproximates the bug pattern and introduces false positives, i.e., error-free configurations that will be misclassified as buggy and will incur some performance impact. Nevertheless, this compression is necessary to fit the patching patterns into an available matcher of smaller size.

To map k patterns into an r-entry matcher, the algorithm first builds a proximity graph. The graph is a clique with k vertices, once for each of the k patterns, and weighted edges connecting the vertices. The weights on the edges are assigned using a variant of the Hamming distance metric. Specifically, we use an additive metric whereby the corresponding bits are compared one to one, and each 0–1 pair contributes 1 to the weight, whereas each $1-x$ or $0-x$ pair contributes 0.5 to the weight. Matching pairs (0–0, 1–1, and $x-x$) do not contribute to the weight. As an example, consider the two patterns $101xx1$ and $1001x1$ shown in Fig. 5. The two leftmost and two rightmost bits of the patterns are identical; thus, they contribute 0 to the weight. Bits 3 of the patterns, on the other hand, form a $0-1$ pair, contributing 1 to the weight, whereas bits 4 form an $x-1$ pair, making the total weight on the edge between these patterns 1.5. The reasoning behind this weighing structure is fairly straightforward: If we were to compact the two patterns connected by an edge, we would have to replace every discording pair ($0-1$, $x-0$, and $x-1$) with an x, basically creating the minimum common pattern that contains both of the initial ones. Matching pairs, however, would retain the values they had in the original patterns. For example, for the two aforementioned patterns $101xx1$ and $1001x1$, the common pattern is $10xxx1$ since we have two discording pairs in the third and fourth bit positions. With this algorithm, each 0–1 pair contributes the same degree of approximation in the resulting entry generated. However, pairs such as $1-x$ or $0-x$ will only have an approximating impact on one of the patterns (the one with the 0 or 1), leaving the other unaffected; hence, the corresponding weight is halved.

An exception to the above metric is a case when one pattern is a subset of another pattern. This is possible because we allow patterns to have don't care bits that essentially represent both 0 and 1 values. In our framework, we set the distance between such proximity graph vertices to -1, guaranteeing that these vertices will be chosen for compression and the more specific pattern will be eliminated from the graph.

Once the proximity graph is built, the two patterns connected by the minimum-weight edge are merged together. If $r \leq k$, the compression is completed; otherwise, the graph is updated

```
 1  PatternCompress(){
 2  for each (pattern i)
 3    for each (pattern j != i) {
 4      if (contains(i, j))
 5        weight(i, j) = -1
 6      else
 7        weight(i, j) =
              compute_distance (i, j) }

 8  while (num_patterns > matcher_lines) {
 9    (i, j) = edge_with_minimum_weight
10    pattern i = merge(pattern i, pattern j)
11    delete pattern j
12    update graph
13    num_patterns -- }}
```

Fig. 6. Pattern-compression algorithm. A proximity graph is initially generated and labeled in lines 2–7. The two closest patterns are merged, and the graph is updated in lines 9–12. The cycle is repeated until the patterns can fit into the fixed size matcher.

using the compressed pattern just generated, instead of the two original ones, and the process is repeated until we are left with a number of patterns that fit in the matcher.

An example of a compression is shown in Fig. 5. Here, for simplicity, we assume that the matcher can only contain two entries and that, initially, there are four bug patterns. After the proximity graph is initially built and the edges are labeled, the algorithm selects the edge with the smallest distance $(D = 1.5)$ and merges patterns $101xx1$ and $1001x1$ connected by it. As was shown before, the resulting pattern is $10xxx1$. When the graph is updated after the first step, it has three vertices and is still too large for the matcher. Note, however, that the pattern that was added $(10xxx1)$ completely overlaps pattern 100001; thus, the edge between them is labeled with distance -1. When the algorithm searches for the edge with the smallest weight for the second step, this edge is selected and the vertex 100001 is eliminated. Compression then terminates since the resulting set of patterns can fit into the two-entry matcher.

Fig. 6 shows a pseudocode for the pattern-compression algorithm. Lines 2–7 generate the initial proximity graph by computing the weights of all the edges either by detecting that vertex i contains vertex j (`contains` function) or by computing the distance using the algorithm described previously (`compute_distance` function). Lines 9–11 select the pair to merge, remove one pattern from the set, and update the graph. The procedure is repeated until we reach the desired number of patterns. Function `merge` in line 10 generates a pattern that is the minimum overapproximation of the two input patterns. The function must first check for containment, in which case it returns the former one. If there is no containment between the two patterns, their approximation is computed by generating an x bit for each nonmatching bit pair. It is worth noting that the performance of the algorithm described could be optimized in several ways, for instance by eliminating all edges with $D = -1$ in the graph at once.

As mentioned before, the compression algorithm generates a set of patterns that overapproximates the number of erroneous configurations. The resulting pattern will still be capable of flagging all the erroneous configurations; however, it will also flag additional correct configurations that have been included by the merging function (false positives). The impact on the overall system will not be one of correctness, but one of performance, particularly if the occurrence of the additional critical control configurations is frequent during a typical execution. We measure the amount of approximation in the matcher's detection ability 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 $1 - \text{false\_positive\_rate}$. Hence, when there is no approximation, the matcher has an ideal specificity of 1; increasing overapproximation produces decreasing specificity values. It is important to note that, by virtue of our design and the pattern-compression algorithm, our system never produces a false negative, i.e., it never fails to identify any of the bug states observable through the selected critical control signals.

D. Processor Recovery

At this point, the set of patterns generated and compressed is issued to the end customers as a patch. We envision this step as being similar to current microcode-patching flow, where a patch for the processor is included into the basic input–output system (BIOS) updates. Updates are distributed by operating system or hardware vendors and are saved in nonvolatile memory on the motherboard. At startup, when BIOS firmware executes, the patches are loaded into the processor by a special loader. FRCL can use an almost-identical mechanism, and we expect FRCL patches to be approximately of the same size of a microcode update ($\sim$2 kB or less). After the patch is loaded at startup into the matcher, the processor starts running. While none of the configurations recorded in the matcher is detected, the activity proceeds normally (we call this mode of operation high-performance). However, when a buggy state is detected, the pipeline is flushed, and the processor is switched to a reliable degraded mode of execution. Fig. 7 shows an example of the execution flow when a bug pattern is matched in an FRCL-equipped processor. In the example, we consider a simple in-order single-issue pipeline, and we further assume that the interaction between a particular pair of instructions INST2 and INST3 triggers a control bug which has been detected and encoded in a pattern already uploaded in the matcher. The figure shows that, when the pattern is detected by the matcher [Fig. 7(a)], the pipeline is flushed [Fig. 7(b)], and the processor is switched to the degraded mode. This mode is formally verified at the design time; hence, we can rely on it to correctly complete the next instruction [Fig. 7(c)]. Finally, the high-performance mode of operation is restored [Fig. 7(d)]. Note that, in a design that was not equipped with the FRCL technology, a problem such as the one just described would probably have required rewriting the compiler software or the microcode related to the instructions to circumvent the bug configuration. Note that it is sufficient to complete only one instruction before reengaging a normal operation since, in the event that the pipeline steps again into an error state, it will, once again, enter the degraded mode to complete the following instruction. On the other hand, a designer may choose to run in a degraded mode for several instructions to guarantee bypassing the bug entirely in a single recovery.

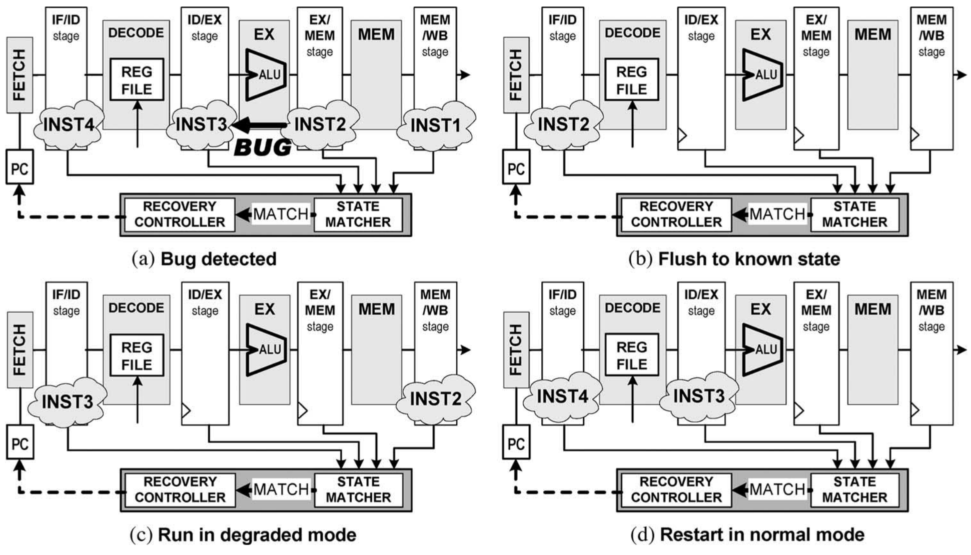

Fig. 7. FRCL in operation. (a) Matcher detects a state associated with a bug described in a preloaded pattern. (b) Pipeline is flushed to a known state. (c) Processor runs in the degraded mode, allowing only one instruction in the pipeline at a time. Degraded mode is formally verified, guaranteeing forward progress and correctness. (d) After offending instructions are bypassed, the processor resumes normal-mode operation.

It should also be noted that, unlike some implementations of the microcode update mechanism, which allow for buggy patches to be loaded [6], our technique cannot introduce new flaws into the processor since our patches only specify when a processor switches to the degraded mode. In the worst case, the processor runs in a degraded mode all the time, with notable performance impacts, but provides correct functionality.

E. Example

We now show the use of FRCL through an example similar to the Intel Celeron bug listed in [1], which we adapt, for simplicity reasons, to a five-stage pipeline. In this example, the processor has a flow that does not always enforce a necessary stall between two successive memory accesses. A stall is required since all memory operations are performed in two cycles: During the first one, the address is placed on the bus, and the data from or to the memory follow during the second cycle. If a memory operation is followed by a nonmemory instruction, they are allowed to proceed back to back since the second operation does not require memory access while advancing through the MEM stage of the pipeline.

In the example, the program that is being run contains a store and a load back to back, which triggers the bug described. The matching logic in this case contains four entries that describe all possible combinations of having two memory instructions in the ID and EX stages of the pipeline. For instance, the first entry matches valid instructions in the ID and EX stages of the pipeline, which are both memory reads. The second entry matches a store in EX, followed by a load in ID, which is triggered during the program execution (Fig. 8). The pipeline is flushed, then the recovery controller restarts the execution at the instruction preceding the store, i.e., the first uncommitted instruction. Note that, in this case, the bug is fully and precisely described by the four patterns loaded in the matcher; thus, no false-positive matches are produced. Moreover, any attempt to compress this set of patterns will introduce false positives, as can be noted by observing the patterns in Fig. 8.

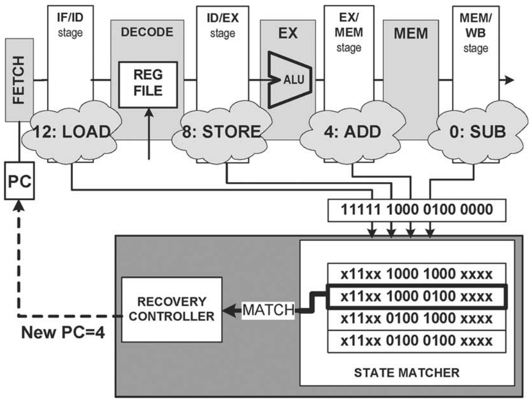

Fig. 8. FRCL for a memory-access bug. Without FRCL, two consecutive memory accesses (8:STORE and 12:LOAD) would be erroneously allowed to proceed back-to-back in the pipeline. When the bug is recognized by the state matcher, the pipeline is flushed, and the execution restarts at the first uncommitted instruction (4:ADD). In the degraded mode, instructions do not go through the pipeline back-to-back, avoiding the bug.

IV. Design Flow

In this section, we describe a design and verification flow that incorporates the FRCL technology. First, we show how the traditional design process needs to be changed to incorporate the FRCL technology and then investigate a formal verification of the degraded mode of operation. Then, we move on to overview control-state selection techniques, including our novel automatic selection algorithm. Finally, we present some insights on incorporating the performance-critical execution into an FRCL-protected design.

A. Overview of the Design Framework

The overall design flow of a component augmented with FRCL is shown in Fig. 9. As mentioned previously, the

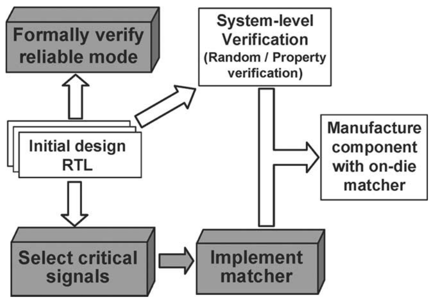
Formally verify reliable mode
System-level Verification (Random / Property verification)
Initial design RTL
Manufacture component with on-die matcher
Select critical signals
Implement matcher

Fig. 9. FRCL design flow: By using the initial RTL, designers formally verify the degraded mode and select the control signals to be monitored by the state matcher. A matcher is incorporated into the final design that is shipped to the end customer.

verification of complex hardware components such as the microprocessors relies today on a variety of formal and simulation-based methods. The deployment of FRCL technology in a processor design requires the addition of two steps to the mainstream design flow. The first step requires us to formally verify the processor when operating in the degraded mode, which is needed by FRCL to recover from the patched design errors. Note that we set up the degraded mode so that instructions are never interacting; hence, verification is greatly simplified. For the most part, this verification effort is reduced to the verification of individual functional blocks, which are already heavily addressed today by formal verification techniques. The system-level verification of the entire processor is still performed using the same mainstream methodology that was used before the deployment of FRCL, typically a mix of random simulation and formal property verification.

The second additional task during the system design is the selection of signals that should become part of the "critical control state." These signals are then routed to a state matcher which was shown in Fig. 4. The number of entries in the matcher is subject to a tradeoff between the total design area and the overall performance of the deployed component since a smaller matcher might require compression and reduce the processor's performance because of the increased false positives.

B. Verification Methodology

In addressing the formal verification of the degraded mode of operation, we exploited a series of optimizations made available by its specific setup. Most of the complex functionality of the processor is disabled in this mode, and only one instruction is allowed in the pipeline at any time, greatly reducing the fraction of the design involved in each individual property proof. To this end, it is important to note that it is not necessary to create a new simplified version of the design. Instead, all of the simplifications are achieved either as a direct consequence of the nature of the input stream—only one instruction is in flight at any one time—or by simply disabling the advanced features through a few configuration bits. For example, modules such as branch predictors and speculative execution units can be turned off with a variant of the "chicken bits," which are control bits used in many design developments to enable and disable features. On the other hand, the control logic responsible for

```
//(1) RTL checker for ADD validity
module add_valid ( INST, valid, fail );
  assign fail = valid & (INST['OPCODE] != 'ADD);
endmodule

//(2) RTL checker for ADD forward progress
module add_forward (clock, reset, IR, valid, ...);
  reg [3:0] count_committed_adds; //saturating
  reg [3:0] clk_cnt; //saturating
  assign fail = (clk_cnt == 4'd5) &
             (count_committed_adds == 4'd0);
endmodule

//(3) RTL checker for ADD semantics
module add_sem (clock, reset, add_in_id,
          add_in_wb, write_dest, write_data, ...);

  reg [63:0] read_a_, read_b_; //operands from RF
  //result register ID read in decode stage
  reg [4:0] dest_id;

  //shows that destination is chosen correctly
  assign fail1 = !((!add_in_wb) || ((add_in_wb)
              & (write_dest == dest_id)));
  // shows that addition is performed properly
  assign fail2 = !((!add_in_wb) || ((add_in_wb)
             & (write_data == read_a_ + read_b_)));
endmodule
```

Fig. 10. RTL checkers to verify the correctness of the ADD instruction with Synopsys' Magellan. Checkers verify (1) the presence of only the valid ADD instruction in flight, (2) forward progress, and (3) correctness of execution.

data forwarding, squashing, and out-of-order execution would be abstracted away by the formal tools due to the fact that only one instruction appears in the pipeline at a time and these blocks are irrelevant. These two major simplifications make the degraded mode operation simple enough for traditional formal verification tools to handle.

In our experiments, we used Magellan from Synopsys to verify both our testbed processor designs. Magellan is a hybrid verification tool that employs several techniques, including formal and directed-random simulation first presented in [19]. Since the instructions are executed independently, we use Magellan to verify the functionality of each instruction in the instruction-set architecture (ISA) one at a time. For each instruction, we wrote assertions in the Verilog hardware design language to specify the expected result. Constraint blocks fixed the instruction's opcode and function field, whereas immediate fields and register identifiers were symbolically generated by Magellan to allow for verification of all possible combinations of these values. An example of checkers written for ADD instruction is shown in Fig. 10. The first module, add_valid, guarantees that only valid instructions, ADDitions in this case, are in execution. The second checker, add_forward, enforces a forward progress by forcing the instruction to complete in a set number of clock cycles. Finally, add_sem enforces the correct semantics for additions by checking that the correct result is written to the register file during the writeback stage. For more complex instructions such as loads and branches, additional checkers are needed to prove that the execution of the operation on the degraded pipelined machine matches exactly the ISA specification.

While we could completely verify the degraded mode for both our testbeds, it should be pointed out that neither could be verified in the high-performance mode because of the much greater complexity involved.

C. Control Signal Selection

A critical aspect of deploying an FRCL is determining which control-state signals are to be monitored by the matcher. On one hand, it would be ideal to monitor all the sequential elements of a design; however, given the amount of control state in complex designs, such approach would be either infeasible or extremely costly. For an FRCL to be practical, the set of critical control signals should be just a handful, selected among any internal net of the design, although this limitation could potentially be the source of false positives at runtime. An example of the impact of a poor signal selection is discussed in Section V, where we describe a bug, r31-forward, used in our experimental evaluation, which describes an incorrect implementation of data forwarding through register 31. In the Alpha ISA, register 31 has a fixed value 0 and, hence, cannot be a reference register for data forwarding. If the critical signal set does not include the register fields of the various instructions in execution, it is impossible to repair this bug without triggering all those configurations which require any type of forwarding, causing an extremely high rate of false positives.

We envision two possible solutions to address this problem. The first and simplest solution is to monitor the destination-register indexes of the instructions at the EX/MEM and MEM/WB stage boundaries by including them in the critical signal set. The downside of this solution is that the critical signal pool would grow and possibly impact the processor's performance; for our in-order experimental testbed, this would be a 30% increase in the signals being monitored. The alternative solution entails including a comparator asserting when a forwarding on register 31 is detected and one additional single bit—the output of the comparator—to the critical set. The additional overhead in this case would be less than the previous alternative. Both approaches would eliminate the false positives for the r31-forward bug and, hence, improve the processor's performance. Thus, a designer using the FRCL approach should keep in mind the possible corner cases such as this and select his critical control pool for a broad range of bugs. A possible approach for this task would be analyzing the previous designs to gain a sense of where bugs have been found.

D. Automatic Signal Selection

Since the critical signal selection is of key importance for FRCL, we have developed a software tool to support a designer in this task. The tool considers the RTL description of the design, and it narrows the candidate pool for the critical control set. It does so by first automatically excluding poor candidates such as wide buses and then by ranking the remaining candidates in decreasing relevance. The rank is computed based on the width of the cone of logic that a signal drives and the number of submodules that they feed into. For example, for the RTL block shown in Fig. 11, the critical state selection tool marks signal A as data, whereas it designates signals B and C as control. However, B will have a higher control signal

```
module example (A, B, C)
    input [64:0] A;
    input B;
    output C;
        assign C = !B & (A == 64'h0);
endmodule
```

A	data
B	Control rank 1
C	Control rank 2

Fig. 11. Example of automatic control selection for a simple module. Signal A is labeled as data because of its width, and signal B is a higher ranked control signal than C since it drives C.

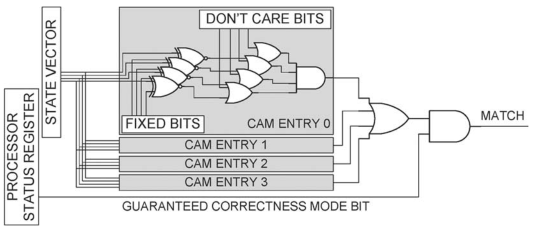

Fig. 12. State matcher with enabled signal asserted or deasserted by the corresponding bit in the processor status register.

ranking since it drives more signals than $C-B$ drives C plus all the nodes that C drives, indicating that it is probably a more important control signal.

When comparing our manually selected critical signal set with the output of the automatic signal selector tool, we noted an 80% overlap. It should be noted that the manual selection was performed by a designer who had full knowledge of the microarchitecture, whereas the automatic selection tool was only analyzing the RTL design.

In Section V, we present an experiment comparing the performance, in terms of specificity (precision of the bug-detection mechanism), of a range of variants of manual and automatic selections. In particular, we looked at the average specificity per signal or the measure of how much each signal is contributing to the precision of the matcher. Solutions with higher average specificity per signal provide a higher specificity, which translates into higher performance, and require less area for fewer signals that need to be routed to the matcher.

E. Performance-Critical Execution

In some systems, the speed of execution may be more critical than its correctness. For example, in real-time systems, it is important to guarantee task completion at a predictable time in order to meet scheduling deadlines. In streaming video applications, the correctness of the color of a particular pixel may also be less crucial than the jitter of the stream. In these situations, our approach that trades off the performance for correctness may be undesirable. For these cases, we propose having an extra bit to enable/disable the matcher (Fig. 12). The matcher-enable bit, however, should only be modifiable in the highest privileged mode of the processor operation to ensure that a user code cannot exploit the design errors for malicious reasons.

V. Experimental Evaluation

In this section, we detail two prototype systems with FRCL support. By using a simulation-based analysis, we examine the error-detection accuracy of FRCL for a number of design-error scenarios and varied state-matcher storage sizes. We also examine different criteria for selecting the control state, including an automatic selection heuristic outlined in Section IV-D. In addition, we examine the area costs of adding this support to simple microprocessors. Finally, we examine the performance impact of the degraded-mode execution to see the extent of error recovery that can be tolerated before the overall program performance is impacted.

A. Experimental Framework

To gauge the benefits and costs of the FRCL, we added this support to two prototype processors. Although experimental in nature, these processors have been already deployed and verified in several research projects. While these prototype processors do not have the complexity of a commercial offering, they are nontrivial robust designs that can provide a realistic basis to evaluate the FRCL solution. For our experiments, we implemented two variants of the state matcher, with four and eight entries, and integrated them into the two baseline processor designs.

The first design is a five-stage in-order pipeline implementing a subset of Alpha ISA with 64-b address/data word and 32-b instructions. The pipeline has forwarding from the MEM and WB stages to ALU and resolves branches in the EX stage. The pipeline utilizes a simple global branch predictor and 256-B direct-mapped instruction and data caches. For this design, we handpicked 26 control bits, which govern the operation of different logic blocks of the pipeline (datapath, forwarding, stalling, etc.), to be monitored by the matcher. These signals were selected through a two-step process: We first analyzed the escaped bugs documented in this paper, which are reported in Section II-A, and then selected those control signals that would have been good indicators of those bugs. This analysis relies on the assumption that future escaped bugs are correlated to past escapes. In addition, in making our selection, we were careful to choose signals which encoded the critical control situations in compact ways: For instance, we chose not to monitor the indexes of source and destination registers of each instruction (which require several bits each), but, instead, we decided to track the occurrence of each data forwarding (only a handful of bits). To limit the monitoring overhead, we also chose not to observe any of the instruction opcode bits that are marched down in each pipeline stage. As detailed in Table I, the majority of the critical control signals were drawn from the ID and EX stages of the pipeline, where the bulk of computation occurs. For example, in the ID stage, we selected some of the output bits of the decoder, which represent, in compact form, what type of operation must be executed, and in the EX stage, we selected the ALU control signals. Although this potentially limited our capability to recognize a buggy state before the instruction is decoded in the ID stage, it allowed us to reduce the number of bits monitored. Note also that, while we chose not to modify the original design in any way, it could be possible to enhance the precision of the error detection by adding minimal additional logic. Examples are the solution to the r31-forward bug described in Section IV-C and also the addition of pipeline latches to propagate more complete information on the instruction being executed through the pipeline, with the result that it would become possible to capture more precisely the specifics of an instruction leading to a bug.

TABLE I
CONTROL-STATE BITS MONITORED IN THE IN-ORDER PIPELINE

Name	Number of bits	Pipeline stage
IF_valid	1	Fetch
ID_valid	1	Decode
EX_valid	1	Execute
MEM_valid	1	Memory
WB_valid	1	Write-back
ID_rd_mem	1	Decode
ID_wr_mem	1	Decode
ID_cond_br	1	Decode
ID_uncond_br	1	Decode
EX_rd_mem	1	Execute
EX_wr_mem	1	Execute
EX_cond_br	1	Execute
EX_uncond_br	1	Execute
EX_ALU_function	5	Execute
EX_br_taken	1	Execute
EX_hazard_source_a	2	Execute
EX_hazard_source_b	2	Execute
MEM_br_taken	1	Memory
MEM_bus_command	2	Memory
Total	**26**	

TABLE II
CONTROL-STATE BITS MONITORED IN THE TWO-WAY SUPERSCALAR OUT-OF-ORDER PIPELINE

Name	Number of bits	Pipeline module
ROB_head_commit	2	Re-order buffer
ROB_head_store_address	1	Re-order buffer
ROB_head_store_data	1	Re-order buffer
ROB_head_load	1	Re-order buffer
ROB_full	1	Re-order buffer
RS_full	1	Reservation Stations
RS_complete	2	Reservation Stations
RS_br_miss	2	Reservation Stations
Issue_branch	2	Rename
Issue	2	Rename
Total	**16**	

The second processor is a much larger out-of-order two-way superscalar pipeline, implementing the same ISA. The core uses Tomasulo's algorithm with register renaming to reorder instruction execution. The design has four reservation stations for each of the functional units and a 32-entry reorder buffer (ROB) 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 memory operations are also performed when a memory instruction reaches the head of the ROB, with a store operation requiring two cycles. The ROB can retire two instructions at a time, unless one is a memory operation or a mispredicted branch. The design also includes 256-B direct-mapped instruction and data caches and a global branch predictor. The signals hand-selected for the critical control pool include signals from the retirement logic in the ROB as well as control signals from the reservation stations and the renaming logic, as reported in Table II.

Similar to the in-order design, no opcodes and instruction addresses were monitored to minimize the number of observed

TABLE III
BUGS INTRODUCED IN IN-ORDER AND OUT-OF-ORDER PIPELINES

Bug	Description
In-order pipeline	
2-mem-ops	Two consecutive memory operations fail
opA-forward-wb	Incorrect forwarding from WB stage on operand A
opA-forward-conf	Incorrect hazard resolution on operand A
2-branch-ops	Two consecutive taken branches fail
store-mem-op	Store followed by another memory operation fails
load-branch	A conditional branch depending on a preceding load fails
mult-branch	A branch following a multiply instruction fails
mult-depend	Multiply followed by a dependent instruction fails
r31-forward	Forwarding on register 31 is done incorrectly
multi-1	*2-mem-ops + opA-forward-wb + opA-forward-conf + 2-branch-ops*
multi-2	*store-mem-op + load-branch + mult-branch*
multi-3	*mult-depend + r31-forward*
multi-4	*2-branch-ops + mult-branch + load-branch*
Out-of-order pipeline	
rob-full-store	Store operation fails when ROB is full
rob-full-mem	Any memory operation fails when ROB is full
double-retire	Double-issue and double-retirement in the same cycle fails
double-retire-full	Retirement of two instruction fails if two non-branch instructions are added to full ROB at the same time
double-mispred	ROB incorrectly flushes the pipeline if two branches are mispredicted at the same time
rs-flush	Reservation stations do not get flushed on a branch mispredict if rs_full signal is asserted
load-data	Loaded data is not forwarded to dependent instructions in the reservation stations
multi-all	All out-of-order bugs combined

signals. The matcher developed for this design was capable of correctly matching the scenarios involving branch misprediction, memory operations, as well as corner cases of operation of the ROB and reservation stations, for example, when they were full and the front-end needed to be stalled. Again, a larger set of signals could be used to gather more detailed information about the state of the machine; however, for this design, the benefit would consist of a shorter recovery time by recognizing the problems earlier on. On the other hand, the ability to precisely identify erroneous configurations would not be improved significantly since errors can still be detected when the instructions reach the head of the ROB.

The processor prototypes were specified in synthesizable Verilog and then synthesized for minimum delay using Synopsys design compiler. This produces a structural Verilog specification of the processor implemented with Artisan standard logic cells in a Taiwan Semiconductor Manufacturing Company 0.18-μm fabrication technology.

For performance analysis, we ran a set of 28 microbenchmark programs designed to fully exercise the processor while providing small code footprints. These programs included branching-logic and memory-interface tests, recursive computation, sorting, and mathematical programs, including integer matrix multiplication and emulation of the floating-point computation. In addition, we ran both of the designs for 100 000 cycles with an interactive stimulus generator StressTest [26] to verify correctness of operation as well as to provide a more diverse stream of instruction combinations.

B. Design Defects

To evaluate the performance of the FRCL solution, we equipped the designs with a matcher block, manually inserted a variety of bugs into our designs, downloaded the appropriate patch to the matcher, and then examined their overall performance. For each bug or set of bugs, we created a variant of the design which included them. In crafting the bugs, we emulated the bugs reported in errata documents that we analyzed in Section II-A. We also strived to target all levels of the design hierarchy. Usually, high-level bugs were the result of bad interactions between instructions in flight. For example, opA-forward-wb breaks forwarding from the WB stage on one operand, and 2-branch-ops prevents two consecutive branching operations from being processed properly under rare circumstances. Medium-level bugs introduced incorrect handling of instruction computations, such as store-mem-op, which causes store operations to fail. Low-level bugs were highly specific scenarios in which an instruction would fail. For example, r31-forward is a bug causing forwarding on register 31 to be performed incorrectly. Finally, the multibugs are the combined bugs, where the state matcher is required to recognize larger collections of bug configurations. For instance, multi-all is a design variant that includes all bugs that we introduced. A summary of the bugs introduced in both of the designs is given in Table III. It can be noted that, even for these simple designs, some of the bugs require a very unique combination of events to occur in order to become visible.

C. Specificity of the Matcher

The control state matcher has the task of identifying when the processor has entered a buggy control state, at which point the processor is switched into a degraded mode that offers reliable execution. In this section, we study the specificity of the state matcher, i.e., its accuracy in entering the degraded mode only when an erroneous configuration is observed.

Figs. 13 and 14 show the specificity of the state matcher for bugs in the in-order and out-of-order processor designs. Recall that the specificity of a bug 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. On the other hand, a matcher with low specificity would overshoot in its analysis and enter the degraded mode more often than necessary. For instance, a specificity of 0.40 indicates that an actual bug was corrected only during 40% of the transitions to a degraded mode, whereas the other 60% were unnecessary. In order to gather a sense of the correlation between specificity and matcher size, we plot our results considering four-entry, eight-entry, and infinite-entry matchers.

It can be noted that, for both processors, many of the bugs can be detected and recovered with a specificity of 1.0 even when using the smallest matcher; thus, no spurious recoveries were initiated. Some combinations of multiple bugs (e.g., multi-1 and multi-2) had low specificities, but when the matcher size was increased, the specificity again reached 1.0. For these combinations of bugs, a four-entry matcher was too small to accurately describe the state space associated with the bugs, but the larger matcher overcame this problem.

Finally, for a few of the bugs, e.g., mult-depend in Fig. 13 and load-data in Fig. 14, even an infinite-size state matcher could not reach the perfect specificity. For these particular bugs, the lack of specificity was not the result of pressure

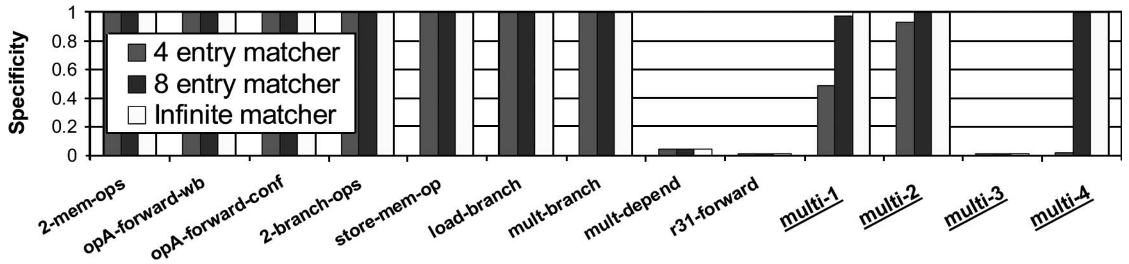

Fig. 13. Specificity of detection for a range of bugs in the in-order pipeline. Low specificity can be due to insufficient critical control monitored by the matcher (bugs mult-depend and r31-forward) or to insufficient size of the matcher (four-entry matcher in bugs multi-1, multi-2, and multi-4).

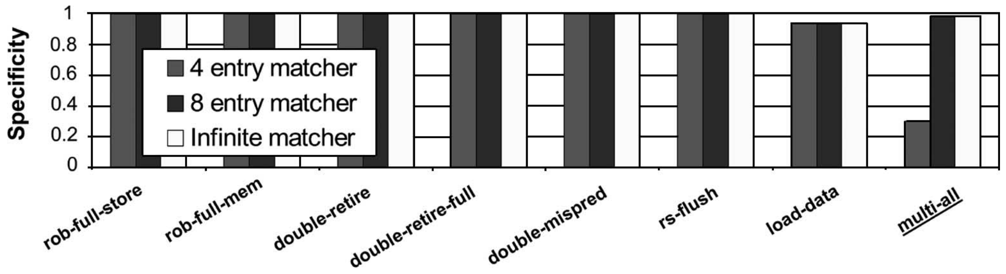

Fig. 14. Specificity of detection for a range of bugs in the out-of-order pipeline. Low specificity can be due to insufficient critical control monitored through the matcher (for instance, load-data) or to insufficient size of the matcher (for instance, the four-entry matcher with multi-1 bug).

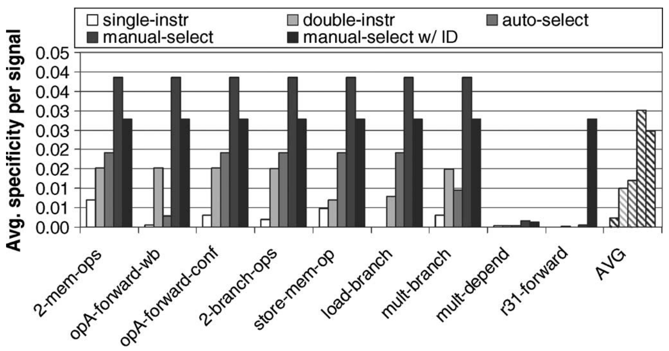

Fig. 15. Average specificity per signal for a range of critical signal sets in the FRCL implementation of the in-order pipeline. In most cases, the manual-select solution achieves the best specificity at lower cost. However, auto-select, based on the automatic selection algorithm in Section IV-C, achieves good results with no effort from the engineering team.

on the matcher but rather of an insufficient access to critical control information, as was described in Section IV-C. Thus, these experiments had to initiate recovery whenever there was a potential error, which lead to the lower specificities.

To evaluate the impact of various critical control signal-selection policies and compare them to the automatic approach described in Section IV-D, we developed a range of FRCL implementations over the in-order pipeline using a different set of critical signals. The results of this analysis are shown in Fig. 15.

In the first configuration developed, single-instr, the critical control consists exclusively of the 32-b instruction being fetched. The second solution, called double-instr, monitors the instructions in the fetch and decode stages (64 instruction bits and 2 valid bits). The third configuration (auto-select) includes all of the signals automatically selected by our heuristic algorithm from Section IV-D for a total of 52 b. For this setup, the automatic selection algorithm was configured to return all RTL signals with nonzero control rank and width less than 16 b. The manual-select implementation exactly corresponds to the one from the experiment in Section V-B, including all the signals listed in Table I; thus, its matcher performance is the same as in the aforementioned experiments. The final configuration, manual-select w/ID, is the same as the manual-select, but it includes ten extra signals to monitor the destination registers in the MEM and WB stages.

Matcher sizes for all of the variants contained enough entries to accommodate even the largest patches; therefore, pattern compression was never required. For each design variant, we developed individual patches for the first nine bugs listed in Table III (all but the multibugs). For each bug and each design variant, we measured the average specificity per signal, i.e., specificity divided by the number of signals in the critical control pool. This measure gives us an intuition on how to select the approach with the best performance/area tradeoff.

As shown in Fig. 15, the manual-select variant produces the best results for most bugs. The manual-select w/ID solution has better specificity than the manual-select solution but at a higher price. Its main advantage is the good result over r31-forward, which is made possible by its tracking destination-register indexes. Note also that the automatic selection algorithm performs quite well, particularly taking into account that this approach does not require any engineering effort.

D. State-Matcher Area and Timing Overheads

Implementing an FRCL solution requires the addition of the critical control matcher logic, i.e., the matcher itself and the recovery controller, which cause an area overhead for the final design. Table IV tabulates the area overheads of a range of FRCL implementations, including the matcher size of four and eight entries built over both the in-order and out-of-order designs and considering 256-B and 64-kB instruction and data

TABLE IV
AREA OVERHEADS AND PROPAGATION DELAYS FOR A RANGE OF FRCL IMPLEMENTATIONS ON THE IN-ORDER AND OUT-OF-ORDER PIPELINES WHEN SYNTHESIZED ON 180-nm TECHNOLOGY

	Critical control state matcher area (% design area)			
	In-order		**Out-of-order**	
	256B	64kB	256B	64kB
4 entry matcher	1.10%	0.01%	0.34%	0.01%
8 entry matcher	2.20%	0.02%	0.68%	0.02%

	Propagation delay of the matcher (ns)	
	In-order (clk=11.5ns)	**Out-of-order** (clk=6.5ns)
4 entry matcher	1.18ns	1.17ns
8 entry matcher	1.43ns	1.21ns

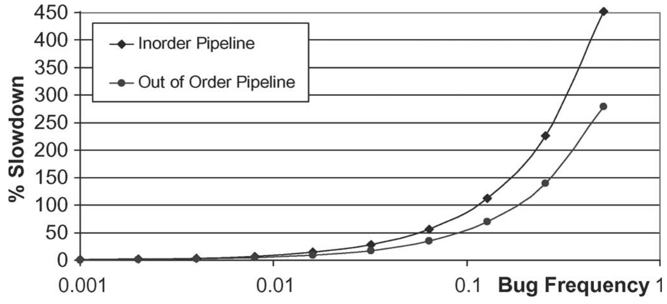

Fig. 16. Impact of recovery on processor performance. FRCL technology incurs less than 5% performance impact as long as the frequency of the bug does not exceed 6 per 1000 cycles in the in-order pipeline and 10 per 1000 cycles in the out-of-order pipeline.

caches. As shown in the table, the overhead of FRCL is uniformly low. Even the larger state matcher with small pipelines and caches (in-order 256-B) results in an overhead of only about 2%. Designs with larger caches and more complex pipelines have an even lower overhead. Given the simplicity of our baseline designs, we would expect the overhead for commercial-grade designs to be even lower. Table IV also presents the propagation delays through the matcher block. Note that all solutions have propagation delays that are well below the clock speed; hence, they do not affect the overall system's performance. Note that the matcher for the out-of-order processor performs faster because it monitors fewer control signals. It should also be pointed out that an FRCL matching is performed in parallel with a normal pipeline operation, and given the observed propagation delays through the matcher, they do not affect the overall design frequency.

E. Performance Impact of Degraded Mode

During recovery, the processor is switched into the degraded mode to execute the next instruction, and then, it is returned to the normal operation. During recovery, only one instruction is permitted to enter the pipeline; thus, instruction-level parallelism is lost, and program performance will suffer accordingly. Fig. 16 shows the performance of the in-order and out-of-order processors as a function of increasing recovery

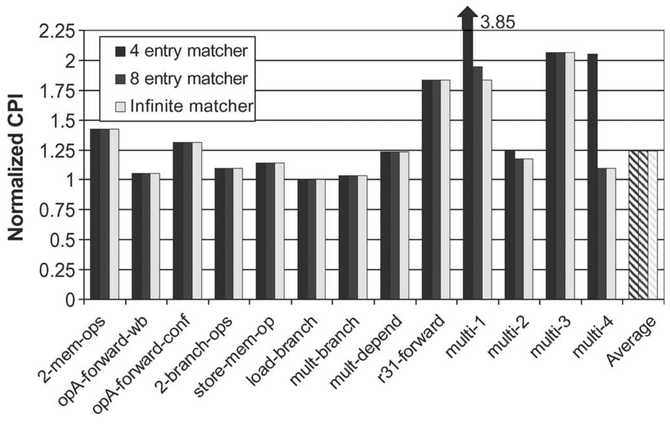

Fig. 17. Normalized CPI for the in-order pipeline. Average CPI increase is computed only over design variants with a single bug.

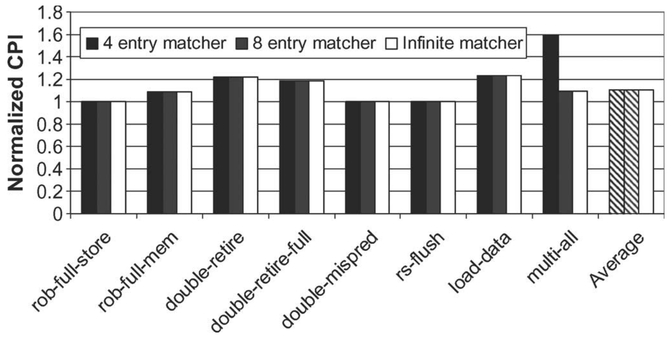

Fig. 18. Normalized CPI for the out-of-order pipeline. Average CPI increase is computed only over design variants with a single bug.

frequency. As shown in the graph, for performance impact to be contained under 5%, the rate of recovery could not exceed 6 per 1000 cycles for the in-order pipeline and 1 per 1000 cycles for the out-of-order pipeline. For a more stringent margin of 2% impact, the recovery rates should not exceed 2 per 1000 and 4 per 1000 cycles for the in-order and the out-of-order processors, respectively. Note that the in-order pipeline suffers more heavily from the frequency of the recovery, as it can be easily derived from its higher sensitivity to instruction latencies.

Finally, Figs. 17 and 18 show the clock cycles per instruction (CPI) of the FRCL-equipped in-order pipeline. The CPI has been normalized to the average CPI achieved when no patch was uploaded on the matcher (hence, the degraded mode was never triggered). By comparison with Fig. 13, it can be noted that low specificity often results in an increased CPI. However, the worst-case scenario (four-entry matcher and multi-1 bug) occurs because of an insufficiently sized matcher and not because of the critical control selection.

VI. CONCLUSION

In this paper, we presented a novel technology called FRCL. We also implemented a microprocessor design solution to detect erroneous control configurations and to recover correct execution through a low-complexity reliable degraded mode. We described a low-cost state-matching mechanism that can detect when to bypass bugs. The technique consistently has an area cost of less than 2%. Moreover, with moderately sized matchers, we can ensure highly accurate detection of bug states in nearly all of our experiments. Finally, we examined the performance impacts of running programs in the degraded mode, and we found that, if recovery frequency is less than ten per 1000 instructions in the out-of-order design and less than six recoveries per 1000 instructions in the in-order design, the performance impact is below 5%. We feel that this paper makes a strong case for FRCL and shows that the approach holds a great promise to ensure against the potential disasters of releasing buggy silicon.

REFERENCES

[1] DDJ Microprocessor Center. [Online]. Available: http://www.x86.org/
[2] *QIfdiv (Enable Pentium FDIV Fix)*. [Online]. Available: http://msdn2.microsoft.com/en_us/library/ms856573.aspx
[3] *Intel(R) StrongARM(R) SA_1100 Microprocessor Specification Update*, Feb. 2000.
[4] *Intel(R) Celeron(R) Processor Specification Update*, 2002.
[5] *Intel(R) Pentium(R) II Processor Invalid Instruction Erratum Overview*, Jul. 2002.
[6] *AMD Athlon (TM) Processor Model 10 Revision Guide*, Oct. 2003.
[7] *Intel(R) Pentium(R) Processor Invalid Instruction Erratum Overview*, Jul. 2004.
[8] *IBM PowerPC 750GX and 750GL RISC Microprocessor Errata Notice*, Jul. 2005.
[9] *Intel(R) Pentium(R) III Processor Specification Update*, May 2005.
[10] *Revision Guide for AMD Athlon(TM) 64 and AMD Opteron(TM) Processors*, Aug. 2005.
[11] A. Allan, D. Edenfeld, J. William, H. Joyner, A. B. Kahng, M. Rodgers, and Y. Zorian, "2001 Technology roadmap for semiconductors," *Computer*, vol. 35, no. 1, pp. 42–53, Jan. 2002.
[12] B. Bentley, "Validating a modern microprocessor," in *Proc. Int. Conf. CAV*, Jul. 2005, pp. 2–4.
[13] B. Bentley and R. Gray, "Validating the Intel Pentium 4 microprocessor," *Intel Technol. J.*, vol. 5, no. 1, pp. 1–8, Feb. 2001.
[14] E. B. Brett, D. P. Hunter, and S. L. Smith, "Moving atom to Windows NT for alpha," *Compaq DIGITAL Tech. J.*, vol. 10, no. 2, Jan. 1999.
[15] D. Van Campenhout, T. Mudge, and J. P. Hayes, "Collection and analysis of microprocessor design errors," *IEEE Des. Test Comput.*, vol. 17, no. 4, pp. 51–60, Oct.–Dec. 2000.
[16] A. Carbine, "Scan mechanism for monitoring the state of internal signals of a VLSI microprocessor chip," U.S. Patent 5 253 255, Nov. 1990.
[17] E. J. McCluskey, "Minimization of Boolean functions," *Bell Syst. Tech. J.*, vol. 6, no. 35, pp. 1417–1444, Nov. 1956.
[18] J. Henry, G. Baker, and C. Parker, "High level language programs run ten times faster in microstore," in *Proc. 13th Annu. Workshop Microprogramming*, 1980, pp. 171–177.
[19] P. H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long, "Smart simulation using collaborative formal and simulation engines," in *Proc. ICCAD*, 2000, pp. 120–126.
[20] K. H. Chang, V. Bertacco, and I. Markov, "Simulation-based bug trace minimization with BMC-based refinement," in *Proc. ICCAD*, Nov. 2005, pp. 1045–1051.
[21] J. K. P. Kevin and J. McGrath, "Microcode patch device and method for patching microcode using match registers and patch routines," U.S. Patent 6 438 664, Oct. 1999.
[22] D. Koncaliev, *Bugs in the Intel Microprocessors*. [Online]. Available: http://www.cs.earlham.edu/ dusko/cs63/
[23] M. D. Goddard and D. S. Christie, "Microcode patching apparatus and method," U.S. Patent 5 796 974, Nov. 1995.
[24] S. Sarangi, S. Narayanasamy, B. Carneal, A. Tiwari, B. Calder, and J. Torrellas, "Patching processor design errors with programmable hardware," *IEEE Micro—Special Issue: Micro's Top Picks from Computer Architecture Conferences*, vol. 27, no. 1, pp. 12–25, Jan./Feb. 2007.
[25] A. Srivastava and A. Eustace, "ATOM: A system for building customized program analysis tools," *ACM SIGPLAN Not.*, vol. 39, no. 4, pp. 528–539, Apr. 2004.
[26] I. Wagner, V. Bertacco, and T. Austin, "StressTest: An automatic approach to test generation via activity monitors," in *Proc. DAC*, 2005, pp. 783–788.
[27] I. Wagner, V. Bertacco, and T. Austin, "Shielding against design flaws with field repairable control logic," in *Proc. DAC*, 2006, pp. 344–347.

Ilya Wagner (S'06) received the B.S. and M.S. degrees in computer engineering from the University of Michigan, Ann Arbor, in 2004 and 2006, respectively, where he is currently working toward the Ph.D. degree at the Advanced Computer Architecture Laboratory, Department of Electrical Engineering and Computer Science.

His research interests include hardware verification and hardware reliability. In summer 2007, he was a Graduate Technical Intern with Intel's Validation Research Laboratory, Hillsboro, OR, researching approaches to pre- and postsilicon validation for multicore processors.

Valeria Bertacco (M'95) received the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1998 and 2003, respectively.

She joined the faculty at the University of Michigan, Ann Harbor, after being with Synopsys for four years as a Lead Developer of Vera and Magellan—the two popular verification tools. She is currently an Assistant Professor with the Department of Electrical Engineering and Computer Science, University of Michigan. Her research interests are in the areas of formal and semiformal design verification with emphasis on full design validation and digital-system reliability.

Dr. Bertacco serves in several program committees, including in the International Conference on Computer-Aided Design and Design Automation and Test in Europe, and she has been leading the effort for the development of the verification section in the International Technology Roadmap for Semiconductors report since 2004.

Todd Austin (M'88) received the Ph.D. degree in computer science from the University of Wisconsin, Madison, in 1996.

He is an Associate Professor with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Harbor. His research interests include computer architecture, compilers, computer-system verification, and performance-analysis tools and techniques. Prior to joining academia, he was a Senior Computer Architect with Intel's Microcomputer Research Laboratories, a product-oriented research laboratory in Hillsboro, OR. He is the first to take credit (but the last to accept blame) for creating the SimpleScalar Tool Set—a collection of computer architecture performance-analysis tools.

Dr. Austin is the recipient of the 2007 ACM Wilkes Award in computer architecture.