Engineering Trust with Semantic Guardians

Ilya Wagner
iwagner@umich.edu

Valeria Bertacco
valeria@umich.edu

Advanced Computer Architecture Lab
University of Michigan, Ann Arbor, MI 48109

ABSTRACT

The ability to guarantee the functional correctness of digital integrated circuits and, in particular, complex microprocessors, is a key task in the production of secure and trusted systems. Unfortunately, this goal remains today an unfulfilled challenge, as even the most straightforward practical designs are released with latent bugs. Patching techniques can repair some of these escaped bugs, however, they often incur a performance overhead, and most importantly, they can only be deployed after an escaped bug has been exposed at the customer site. In this paper we present a novel approach to guaranteeing correct system operation by deploying a *semantic guardian* component. The semantic guardian is an additional control logic block which is included in the design, and can switch the microprocessor's mode of operation from its *normal*, high-performance but error-prone mode, to a a secure, formally verified *safe mode*, guaranteing that the execution will be functionally correct. We explore several frameworks where a selective use of the safe mode can enhance the overall functional correctness of a processor. Additionally, we observe through experimentation that semantic guardians facilitate the trade-off between the design validation effort and the performance and area cost of the final secure product. The experimental results show that the area cost and performance overheads of a semantic guardian can be as small as 3.5% and 5%, respectively.

1. INTRODUCTION

System security and robustness has become a key concern in many electronic computing systems. This concern is most pronounced in military and medical systems, where lives depend on a system operating correctly all of the time, free of inadvertent or malicious faults. Today, nearly all of the research into system security is focused on software correctness, since a system is most likely to be compromised through an exploitation of a software bug. However, it is also possible for a system to become compromised because of hardware bugs. For example, the Intel Pentium F00F bug [3] consisted of an invalid instruction sequence that would lock up a processor. A hardware bug such as this could be exploited remotely to implement denial-of-service attacks on systems based on this processor. This attack could be perpetrated even on systems running completely correct software, since it relies exclusively on an underlying hardware flaw. Unfortunately, as of today, even the simpler commercial designs are released with latent bugs, which are often exposed after the product has reached the customer, with consequences that range from simple inclusion to an *errata* document [4, 1, 2] to the unfortunate extreme of a product recall [13]. While the past few decades have witnessed significant efforts to improve the hardware verification methodology, these efforts have been far outstripped by the massive complexity increase in modern digital designs, so that today only a vanishingly small fraction of a design's state space can be validated before the device is manufactured. To limit the exposure to a potential recall, hardware designers have recently explored a few solutions, including microcode patching [11, 10] and field-repairable control logic [12]. Unfortunately, these mechanisms often have limited scope in terms of the design errors that they can overcome and can incur high performance overhead. Most importantly, they can be deployed only *after* an escaped bug has manifested itself and has been identified, which is often too late for any safety-critical application.

1.1 Contributions

The main contribution of this paper is a novel approach that makes a step towards achieving complete functional correctness in hardware designs such as microprocessors. In this framework a hardware component called a *semantic guardian* is automatically synthesized, based on validation coverage data, and it is included together with the target system's hardware. At run time, the guardian monitors a subset of the design's internal nodes, and when a non-validated configuration is encountered it switches the device into a *safe mode*. The safe mode is an extremely stripped down and lower performance mode of operation of the design, simple enough to be formally verified. By means of the semantic guardian and the safe mode we can make sure that the device is always operating in a verified configuration, at least with respect to the observability nodes of the semantic guardian. In addition, it is possible to derive a trade-off between the effort spent in design-time validation and the runtime performance cost due to the fraction of time spent in the safe mode. We show that our approach is orthogonal to traditional hardware patching and present examples on how it can be combined with previous solutions. Our experiments on an in-order and an out-of-order processor pipelines show that this approach can introduce as little as 3.5% area overhead and protect a complex microprocessor from control bugs with just 5% performance overhead.

The remainder of the paper is organized as follows: In Section 2 we review previous approaches to trusted hardware design and hardware patching. In Section 3 we present the development and execution framework for our solution. Section 4 illustrates how semantic guardians can be used in synergy with hardware patching and Section 5 provides a case study of a bus master device. The last three sections discuss limitations, experimental results and conclusions.

2. PRIOR WORK

Traditional verification of hardware designs relies mostly on simulation and constrained-random testbenches, which exercise a design with streams of stimuli resembling real-life applications. These approaches, however, cannot provide the strong guarantees of design correctness which are often required for safety-critical devices. Alternatively, formal methods, such as symbolic simulators, model checkers and theorem provers, attempt to prove mathematically that

978-3-9810801-2-4/DATE07 © 2007 EDAA

a device behaves correctly under any operating condition. Unfortunately, so far these methods had only partial success in microprocessor verification. For example, even a simple safety critical processor such as the VIPER [8] had to be validated with "intelligent exhaustive simulation", since it was too complex for formal tools. In 1995 Ho *et al.* [9] attempted to verify a processor by generating all possible control logic transactions. Although this proved effective for a simple RISC processor, this approach doesn't seem to scale to handle today's commercial designs, which include complex features such as wide issue, out-of-order execution, simultaneous multi-threading, and complex memory hierarchies. As an example, the formal verification effort on a mainstream processor such as the Intel Pentium 4 was only limited to floating point units and decoder logic [6]. Thus, due to the limited scalability of formal techniques, simulation-based approaches still remain the workhorse of the verification effort in industry.

An alternative approach to guaranteeing microprocessor correctness at run-time was proposed by Austin in [5]. This work proposes a solution where a simple trusted processor checks the results computed by a high-performance core, which has been validated, but could not be formally verified. If the results of the high-end core are erroneous, the simple pipeline corrects them on-the-fly. In contrast, in our solution we keep track of control states validated at design-time, and only use our safety mechanism for the not-yet-validated configurations. In addition, we can correct the execution using the same hardware of the original system, by simply stripping it down to its barebone functionality. The barebone version is sufficiently simplified that we can formally verify its correctness at design-time. In addition,microprocessor design houses have recently started to employ techniques to correct design flaws in the field. The most common of these approaches is microcode patching [11, 10], which modifies the semantics of execution of individual instructions by providing an alternative translation of the instructions to micro-operations. This allows to bypass faulty situations in a way that is transparent to the software applications running on the processor. In our earlier work, we proposed an approach called field-repairable control logic, which allows to patch hardware at the control level [12]. This framework relies on the manufacturer issuing a patch after an escaped bug is identified and reported. The patch encodes all the configurations in which the bug manifests, and it is loaded into a specialized memory in the processor as start-up. When any of the "buggy" configurations are observed at runtime, a recovery mechanism is triggered. Although, these approaches can correct escaped bugs in relatively cheaply and incur moderate overheads, they still can tackle bugs only after they have manifested themselves and have been reported. Hence, they provide no guarantees that escaped, but undiscovered, bugs cannot compromise device's robustness.

3. TRUSTED HARDWARE DESIGN

The design flow we envision for our solution is shown in Figure 1 and is similar to traditional design and verification flows. We require that the design team formally verifies the units that provide the key functionality of the device, *i.e.* the *safe mode* operation. For example, core units required for a processor's safe mode include the datapath blocks, while forwarding logic, pipelining, branch prediction, *etc.* are performance enhancement units needed only in *normal mode*. Note that, in safe mode the device can perform all its necessary functions, but only at a baseline performance. Moreover, since the safe mode provides only barebone functionalities, it is simple enough to be tackled by modern formal verification tools. The normal mode of operation, including all the performance enhancements, is also validated extensively (as it is common practice today) through a mix of semi-formal and simulation tools. The main purpose of this effort is to verify that the most typical, and frequently occurring, operation scenarios are designed correctly.

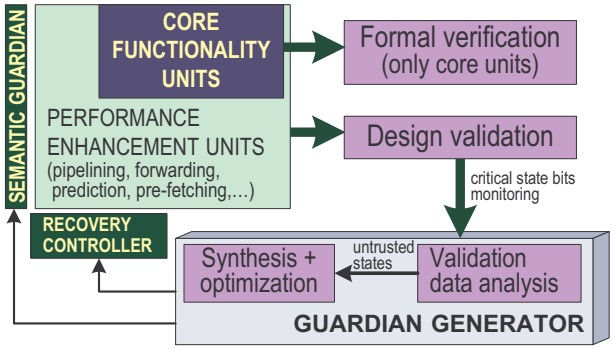

Figure 1: Trusted hardware design flow. The safe mode is verified thoroughly with formal tools, while the normal mode is validated with focus on the most common functionality. A semantic guardian is then automatically generated and manufactured with the design. The guardian, together with a recovery controller switches the design into the safe mode when any non-validated scenario is observed at runtime.

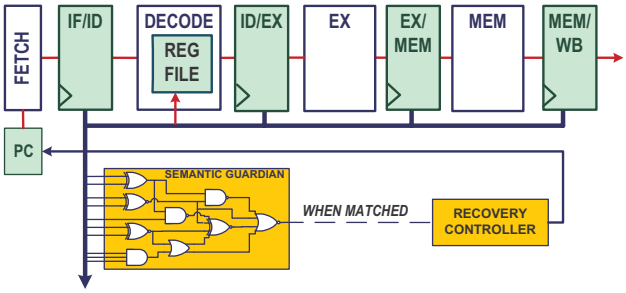

Figure 2: Trusted execution model with a semantic guardian. When an un-trusted state in the pipeline's operation is observed by the guardian, the recovery controller switches the processor into a safe single-cycle, single-instruction mode (*safe mode*). Once the un-validated configuration is bypassed, the device transitions back to normal mode.

An important step in this process is the identification of the signals which represent the critical internal state of the design. The values observed on these signals during validation are monitored by the guardian generator. The guardian generator tracks which configurations of the design have been explored during the validation, and considers those to be *trusted* states, since it assumes that the verification team has validated the behavior of the device for those states. All the configurations that were not validated are considered *un-trusted*. Afterwards, the guardian generator creates a *semantic guardian*, as a combinational logic block, which flags all the un-trusted states. In Section 3.2 we elaborate on the details of the operation of the guardian generator and present a specialized guardian synthesis and optimization flow. The generator also creates a recovery controller, which is connected to the output of the semantic guardian. The controller has the responsibility of switching the design into the safe mode whenever the guardian flags an un-trusted

state. It does so by disabling all design's blocks except for the core functionality units. For example, for the pipelined processor shown in Figure 2, the guardian monitors the critical state set, and, when an un-trusted state is encountered, it signals the recovery controller. The controller squashes all unfinished instructions and restarts execution from the first un-committed operation, forcing the safe mode. When the un-trusted state is bypassed in safe mode, the recovery controller restores normal mode operation.

3.1 Critical signal selection

As mentioned in the introduction, today's designs feature highly complex control blocks and multiple communication interfaces, which are prone to design errors. The resulting system is often so complex that modern formal verification techniques cannot fully guarantee its proper operation in all cases. Therefore, the selection of critical signals to be monitored by the semantic guardian should focus on the key control signals within these complex performance-enhancement blocks. For example, a module controlling data forwarding in a microprocessor pipeline might take IDs of a source and destination registers of instructions from different pipeline stages and output forwarding bus control signals. Selecting these critical nodes to be monitored by the guardian allows to prevent potential escaped errors in the forwarding mechanism. Similarly, control inputs to critical blocks such as computational units, bus interfaces, and memory elements in the design's control FSMs are priority candidates for guardian monitoring.

3.2 The guardian generator

The guardian generator monitors all the configurations of a device explored during validation, with respect to the selected critical signals. For each distinct configuration observed, a confidence metric is calculated. This allows a designer to distinguish the configurations that can be trusted to operate correctly from the ones that have not been covered, or have been covered insufficiently. The un-trusted configurations are then mapped into a combinational logic block, which is synthesized and optimized to generate a semantic guardian, whose output indicates the observation of an un-trusted state.

To optimize area and propagation delay of the semantic guardian block, we introduced a range of heuristic optimizations. First, we synthesize both our un-trusted set and its complement, and then we select the smaller circuit. Then, if any of the un-trusted configurations has been proven unreachable at design time, we use it as don't care in optimizing the guardian design. Additionally, designers might choose to trade off the specificity, or accuracy, of the semantic guardian for better physical parameters of the matching circuit. In this scenario, some of the trusted states might be re-labeled as un-trusted and included in the set, which the guardian will flag. This operation has the potential to increase the effectiveness of the optimization algorithm, and therefore generate a smaller and faster semantic guardian. On the other hand may cause false positives in the guardian detection mechanism, hence, it is important to take the frequency of occurrence of a state into consideration when performing this type of optimization. Section 7 investigates the trade-off between guardian accuracy and area/performance.

In the experiments described in this work we assume that errors can only arise due to the device entering a state leading to erroneous computation. However, our approach can also be used in a framework where errors are modeled as erroneous transitions between design states. In this case, transitions represented as pairs of states (source and sink) must be tracked during validation and encoded in the guardian. Although this approach allows to pinpoint bugs more precisely, the number of transitions, even with respect to our critical signal abstraction, could be extremely large and, therefore, the majority of them would be classified as untrusted even after significant validation effort.

4. SEMANTIC GUARDIAN AND PATCHING

One important aspect of the approach presented here is that it is orthogonal to traditional hardware patching methods. Since the semantic guardian only observes the internal state of a microprocessor, the microcode or software instructions that are executed on it are irrelevant to the robustness of the system. This enables the design team to implement a guardian for a patching mechanism itself, for example, to check if a microcode patch is addressed correctly.

Additionally, it is possible to envision a design where both a guardian and a hardware patching mechanism work in synergy. For instance, the patching mechanism could be used to tune the processor to perform trusted activities more frequently. In the case of a patching mechanism such as the one of [12], where a specialized memory flags specific critical states, the patching memory can be used to overrule the semantic guardian's decision for selected configurations. The configurations of choice would be those that have been proven correct after the design's release, resulting in an overall performance boost for the processor.

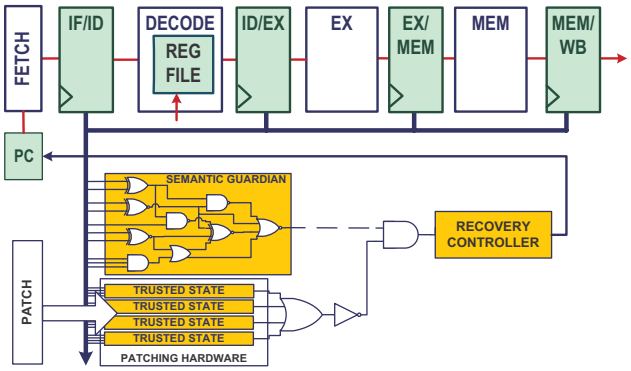

Figure 3: Semantic guardian and patching hardware working in synergy. A patch extending the set of trusted configurations is uploaded to the system deployed in the field. A match in the guardian can be overwritten by the patching hardware as a trusted state, avoiding the transition to safe mode.

This scenario is illustrated in Figure 3: the hardware designer continues the device validation after the release. If and when an un-trusted configuration becomes sufficiently validated, it is encoded and uploaded onto the specialized processor memory at runtime. At this point, if the configuration is ever flagged by the guardian, the patching mechanism overwrites the decision, preventing the transition to safe mode. Therefore, this approach allows the design team to expand the set of trusted configurations even after the device has been manufactured and shipped.

5. CASE STUDY

In this section we present a case study that illustrates the proposed approach. The target device for our case study is a bus master which accepts requests from several devices

with different priorities and allows transactions on the bus to be pipelined. For instance, one device can be receiving a grant and preparing for a transaction, while another can be transmitting the address, while a third one can be sending data. Note that pipelining improves the bus master performance but introduces a fair amount of complexity. We can simplify the system by allowing only one transaction in flight at a time: this can be our safe mode. The simple safe mode can be verified with formal methods, while the high-throughput, pipelined mode, may be too complex and has to be validated through simulation. A possible execution scenario involves a transaction being deferred while receiving high-priority request. If such situation could not be generated and validated at design-time, we would label it as un-trusted, and the corresponding critical signals values would be included in the semantic guardian.

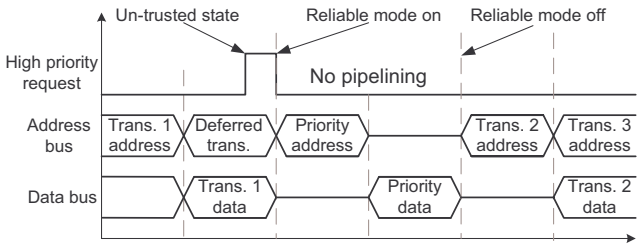

Figure 4: Trusted hardware paradigm. A pipelined bus master design enters an un-trusted state when a high-priority access is requested during a deferred transaction. The transaction is terminated and the bus is granted to the high-priority device in non-pipelined safe mode. After the high-priority transaction is complete, the high-performance mode of operation is resumed.

When the master operates, the unverified scenario described above would produce a match forcing the bus into the non-pipelined safe mode, which allows the high-priority transaction to go through as soon as possible. This event sequence is illustrated in Figure 4. Although the master might work correctly in this situation even in high-throughput mode, it is also likely that the control logic for this complex guarantee contains a bug, especially since this functionality was not validated. By introducing a semantic guardian we make sure that no bugs, with respect to the abstract states observed by the guardian, can manifest themselves, even in rare and unverified cases.

6. LIMITATIONS AND EXTENSIONS

It should be noted that there are a few factors which may limit the usefulness of the proposed approach. First, the critical signals' selection may impact the ability to detect potential errors. In fact, these signals represents a design's abstraction and if an error is triggered by signals which are not being monitored, then it would not be detected by our approach. Hence, in general a poor selection of the critical signals can obliterate the protection that our approach offers. A possible technique to make the solution to overcome this type of problems is to create a series of semantic guardians, each observing different portions of the activity, and all connected to the recovery controller.

A second limitation of the method derives from the fact that some states in the complete system may be labeled as trusted, although they are not fully validated. The decision for the labeling is made by the verification team, based on coverage and thoroughness of the validation, and it is not guaranteed to be always correct. Thus, a configuration might be mistakenly labeled as trusted, and safe mode would not be triggered when it occurs. The alternative approach mentioned in Section 3.2, where transitions between states are monitored and labeled as trusted, can provide a somewhat finer control in this case.

7. EXPERIMENTAL RESULTS

The following section presents the details of our experimental testbed, as well as the results of the analytical experiments with semantic guardian generation flow, compression and optimization, performance/area tradeoff, and, finally, bug resilience of our approach.

7.1 Evaluation Platform

For our experiments we used two processor cores running a subset of the Alpha instruction-set. The first core contained an in-order five-stage single-issue pipeline, while the second design, that we used in the last experiment only, has a two-way superscalar out-of-order pipeline with renaming to the re-order buffer. Both designs included small direct-mapped instruction and data caches, and a global branch predictor unit. The safe mode of operation for both designs was a non-pipelined single-issue mode of operation where caches, branch prediction, and speculative execution were disabled. Since forwarding, stalling and speculative execution logic were unused in this mode, both designs were simple enough to be formally verified with Synopsys Magellan for all instruction types. Using the techniques described in [12] we selected 26 control signals in the in-order pipeline and 16 signals in the out-of-order core to be observed during validation by the guardian generator. For semantic guardian's synthesis and optimization we used a combinations of several techniques, including different configuration of Espresso [7] computing the ON- or OFF-set of the combinational function. We also developed a proprietary heuristic which progressively collapses pairs of states with Hamming distance of 1. Also, as mentioned in Section 3.2, if the un-trusted set encompassed more than 50% of the total state space, the trusted set was used instead to generate the guardian. Finally, we developed a script which considers the output of Espresso or of our synthesis heuristic and produces an register-transfer level description of the guardian circuit, which is then synthesized with Synopsys' Design Compiler and mapped to TSMC 0.18 and 0.13 μm libraries.

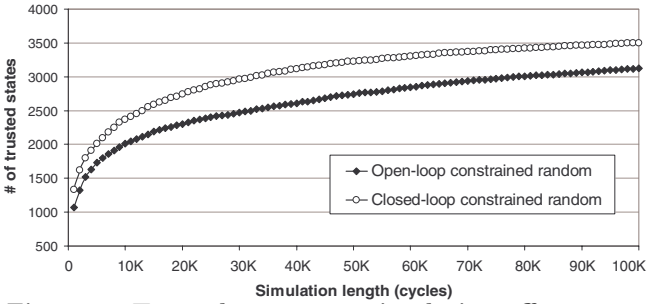

Figure 5: Trusted states vs. simulation effort. With increasing number of simulation cycles more states can be checked and labeled as trusted. Moreover, a more complex closed-loop verification technique performs better then simple constrained random approach.

7.2 Semantic Guardian Generation Results

In our first experiment we calculated the number of distinct critical signal values combinations (trusted states) observed during the validation of the normal mode of opera-

Table 1: Area and propagation delay of the semantic guardian generated with different optimizations.

Optimization method	#ON-set	#DC-set	TSMC 0.18μm area mm^2	TSMC 0.18μm delay ns	TSMC 0.13μm area mm^2	TSMC 0.13μm delay ns
Without Design Compiler optimization						
No optimization	52104	0	0.0186	4.07	0.0099	3.78
Espresso+	2455	11091	0.0194	2.79	0.0101	**2.42**
Espresso-	10586	1556	0.026	4.13	0.0144	3.24
Compaction	18137	843	**0.0121**	**2.76**	**0.0062**	2.47
Comp & Espr+	2449	11097	0.0196	2.78	0.0101	2.46
Comp & Espr-	10586	1556	0.0252	4.44	0.0143	3.23
With Design Compiler optimization						
No optimization	52104	0	0.0268	1.55	0.0168	1.28
Espresso+	2455	11091	0.0212	1.14	0.0171	**0.93**
Espresso-	10586	1556	0.0324	1.59	0.0264	1.27
Compaction	18137	843	**0.0173**	1.19	**0.0143**	1.01
Comp & Espr+	2449	11097	0.0246	**1.08**	0.0160	0.96
Comp & Espr-	10586	1556	0.0316	1.56	0.0269	1.3

tion. To produce stimuli we used both an open-loop and closed-loop constrained random generators. In our framework, a state was considered trusted if it is observed at least once during the simulation, since the signals we selected thoroughly describe the behavior of the processor control FSM. The results of this experiment are shown in Figure 5. It can be observed that, with increasing simulation length, *i.e.* increasing validation effort, the number of states labeled as trusted increases, leveling off at the far right of the graph. Note that we could not perform a formal verification of this design and, therefore, cannot estimate the fraction of the overall reachable state-space covered during simulation.

In the second experiment we analyzed various compression techniques for best area-delay parameters of the semantic guardian matching logic. The results of this study are presented in Table 1. The columns list the compression technique, number of set bits and don't care bits in the in the Boolean function for the guardian and area in mm^2 and propagation delay through the semantic guardian in ns. Espresso+ and Espresso- indicate the circuit was obtained optimizing the ON-set or the OFF-set with Espresso. Compaction is our heuristic optimization technique described above, and the last are combination solutions. The top half of the table shows results with no additional optimizations by Design Compiler, while the bottom half shows the best possible circuit that Design Compiler was able to produce with the most narrow timing and area constraints.

In general, we found that with the more stringent area and timing constraints, Design Compiler gives higher priority to delay optimization and produces a significantly faster guardian. We also noted that Espresso generating the ON set (Espresso+) or Espresso in combination with our compaction approach (Comp & Espr+) generated the circuits with the smallest delay, at a tolerable area penalty. For comparison, the area of the in-order core design, excluding caches, in 0.18 μm technology was 0.5 mm^2. Thus, with Design Compiler optimization the best guardian can incur as little as 3.5% are overhead in the design.

In the third experiment we investigated the possibility of generating a smaller and faster circuit by re-labeling some of the trusted states as un-trusted. Note that in this experiment, the semantic guardian was generated from the trusted set, therefore, when trusted, but rare, states are removed from the set, the guardian becomes smaller. This effect is amplified due to better compressibility of the set with rare states removed. In other words, Espresso and our compaction heuristic were capable to achieve a more effective

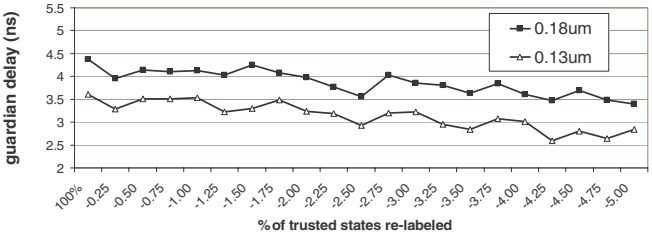

Figure 6: Delay impact of trust re-labeling.

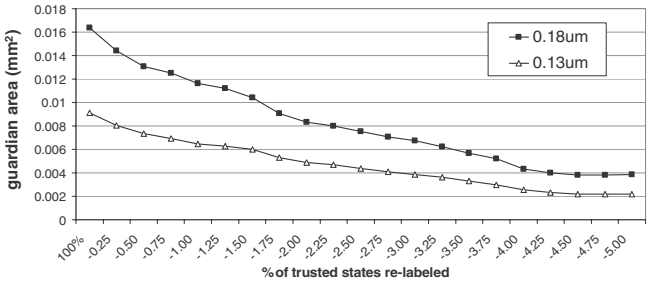

Figure 7: Area impact of trust re-labeling.

simplification without these states. In addition, it is possible that when these states were removed, more DC combinations became available, further simplifying the guardian design.

For this experiment, each state observed in validation is labeled with its observation frequency, that is, the number of times the state has been seen, divided by the total number of simulation cycles. The baseline matcher included all trusted states and hence its cumulative observation frequency was equal to 100%. Then we grouped the states in clusters whose cumulative observation frequency was 0.25%, starting from the lowest frequency states. Each of these clusters was then progressively removed, one at a time, from the pool used to generate the semantic guardian, and we created a new guardian each time without timing or area constraints. The results of this experiment are presented in Figures 6 and 7, where the X-axis shows the trusted-set size reduction in percent of observation frequency. It can be observed from the diagrams that there is a definite reduction in area for smaller trusted sets, enabling up to a 4x area reduction for just a 5% penalty in observation frequency. This trend can also be observed in the matching delay. The trust re-labeling algorithm allows for the designer to trade off the precision of the matcher to achieve improved area and delay for the guardian circuit.

7.3 Performance Evaluation

To evaluate the performance drop due to safe mode operation we augmented the in-order processor core described earlier with semantic guardians created from closed-loop constrained random simulations of different lengths, and, thus, having trusted sets with different sizes. We evaluated the processor on a set of benchmarks stressing different aspects of the pipeline operation, and we measured the average CPI in each case. As shown in Figure 8, guardians created with less verification effort incur a significant slowdown, however, the curve levels off after a while at a 5% CPI overhead. Therefore, the optimal semantic guardian in this case is not the most complete circuit, but rather a smaller guardian that provides tolerable CPI penalty. In this experiment the baseline CPI for the in-order design running these benchmarks was 1.48, and the safe mode was invoked roughly once for every 200 instructions.

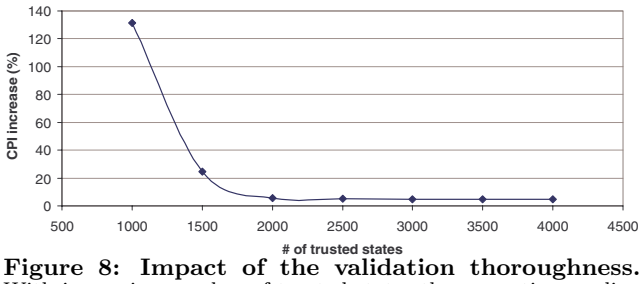

Figure 8: Impact of the validation thoroughness. With increasing number of trusted states the semantic guardian imposes smaller CPI overhead.

Table 2: CPI overhead for different design errors

Bug name	Exposing sequence	CPI
In-order pipeline		
Ideal/Worst		**1.48/5.0**
br+br	2 consecutive branches	**caught**
ubr+cbr	Cond. branch after uncond. branch	**caught**
mem+mem	2 memory accesses back-to-back	1.92
forwarding1	Forwarding source A from WB	1.74
forwarding2	Forwarding source A from MEM/WB	2.10
store+mem	Store followed by another memory access	1.67
load+branch	Load followed by a dependent branch	1.67
Average		**1.82**
Out-of-order pipeline		
Ideal/Worst		**2.25/7.0**
2retire+2issue	Double issue and double retire	**caught**
rob+store	Store at the head of full ROB	2.99
rob+memory	Memory access at the head of full ROB	3.09
non_br_issue	2 non-branch instr. issued when 2 retire	**caught**
2mispred	2 simultaneous branch mispredictions	3.27
mispredict+rs	branch misprediction if RS's are full	2.99
Average		**3.09**

In our final experiment we tested the actual robustness of our approach, by inserting seven distinct errors, one at a time, into the control logic of the in-order pipeline. The errors were based on the publicly available errata of modern microprocessors adapted for our design. All of the bugs involved complex interactions of multiple instructions in the pipeline; note that these scenarios did not exist in the formally verified safe mode, when only one instruction was allowed in the pipeline at any time. Two of the errors were caught during the validation process of the complete system (we used a closed-loop constrained-random simulator). Five of the errors, on the other hand, escaped into the design, because the simulator could not expose them. Nevertheless, the semantic guardians produced during the validation could completely protected the system from these errors, and all 28 benchmarks terminated correctly when running on buggy processors with semantic guardians. We conducted a similar experiment with the out-of-order pipeline: out of six inserted bugs, two are caught in validation, while other escaped. Nevertheless, all tests terminated correctly. Details of this experiment are presented in Table 2. The first column of the table lists the bug name, the second gives a short description of the flaw, and the last column shows if the bug was caught during validation or escaped and caused any overhead because of the semantic guardians.

8. CONCLUSIONS

In this paper we presented a novel approach to trusted hardware design that is orthogonal to traditional hardware patching techniques. Our approach makes use of a *semantic guardian* circuit that is generated automatically based on an analysis of the validation effort. The guardian ensures that at runtime the system always stays in a verified state, which is either a trusted state of the normal operation mode or a formally verified safe mode configuration. The safe mode is invoked by the guardian every time an un-trusted state is encountered. Our approach allows to directly trade off verification effort, area, and performance of the system and successfully combat escaped design errors as shown by our experimental results. We also investigated several optimization techniques that allow for a significant reduction of area and propagation delay of the semantic guardian circuit. In addition, we investigated the tradeoffs between verification effort, area and performance that our methodology offers. The results of this study show that a semantic guardian protecting from a variety of highly complex bugs can incur as little as 3.5% area overhead and have a performance penalty of approximately 5%. Finally, we discussed several approaches of how semantic guardians can be used jointly with traditional hardware patching techniques to increase the level of system robustness or to enable validation to continue even after the system is released.

9. REFERENCES

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

[2] *Intel(R) 8xC251Sx Specification Update*, Nov. 2001.

[3] Intel(R) Pentium(R) Processor Invalid Instruction Erratum Overview, July 2004. `www.intel.com/support/processors/pentium/sb/cs-013151.htm`.

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

[5] T. Austin. DIVA: A Dynamic Approach to Microprocessor Verification, May 2000.

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

[7] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[8] B. Brock and W. A. Hunt. Report on the Formal Specification and Partial Verification of the VIPER Microprocessor. In *Compass '91: 6th Annual Conference on Computer Assurance*, pages 91–98, Gaithersburg, Maryland, 1991. National Institute of Standards and Technology.

[9] R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill. Architecture validation for processors. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 404–413. ACM Press, 1995.

[10] 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.

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

[12] I. Wagner, V. Bertacco, and T. Austin. Shielding Against Design Flaws with Field Repairable Control Logic. In *DAC, Proceedings of Design Automation Conference*, 2006.

[13] A. Wolfe. Intel "bug disaster". *EE Times*, July 97.