Investigating the Limitations of PVF for Realistic Program Vulnerability Assessment

Björn Döbel[1], Horst Schirmeier[2], and Michael Engel[2]

[1] TU Dresden, Operating Systems Group
[2] Technische Universität Dortmund, Department of Computer Science 12

Abstract. From a software developer's perspective, fault injection (FI) is the most complete way of evaluating the sensitivity of a program against hardware errors. Unfortunately, FI campaigns require a substantial investment of both, time and computing resources, making their application infeasible in many cases.
Program Vulnerability Factor (PVF) analysis has been proposed as an alternative for estimating software vulnerability. In this paper we present PVF/x86, a tool for computing the PVF for x86 programs. We validate the use of PVF analysis by running PVF/x86 on an image decoder application and compare the results to those obtained with a state-of-the-art FI framework. We identify weak spots of PVF analysis and outline ideas for addressing those points.

1 Introduction

As the likelihood of hardware errors increases with each technology generation, the implementation of measures dealing with such errors in hardware and software is gaining importance. While reliability has been a research topic for computer architecture researchers for a long time [8], we believe that it is now also becoming an important issue for software developers.

Developers today use a variety of metrics when it comes to evaluating their code, including code complexity, execution time, and test coverage. Measuring important metrics is an integral part of the development and quality assurance cycle. Future software developers will also require metrics to evaluate the reliability of their code with respect to different types of hardware errors.

Our goal is to provide developers with tools to perform reliability analysis. Given a fault model, such tools should be able to answer a variety of questions, such as:

1. *Which parts of a program are most vulnerable to faults?* Development time is a scarce resource and therefore developers would like to focus their efforts on those parts of a program that are most vulnerable.
2. *Given two implementations A and B of a feature, which one is less vulnerable to faults?* Answering this question may aid design decisions. Furthermore, it will allow constant quality monitoring by giving immediate feedback about how the last patch modified the program's vulnerability.

3. *How do compiler optimizations impact vulnerability?* Compilers may use a reliability metric as an optimization criterion. Furthermore, we would like this metric to be able to estimate how the application of software fault tolerance methods, such as replication [2] or encoded processing [10] influences vulnerability.

A common practice today is to use fault-injection (FI) experiments to evaluate the vulnerability of software with respect to a given fault model. Unfortunately, these campaigns consume a lot of time. Hari et al. describe search space optimizations that reduce experimentation time so that their given set of applications "can be simulated in approximately 11 days on a cluster of 200 cores." [4] This is too long for a single software developer to estimate the effects his last code change had on the reliability of an application.

We are not the first ones to observe the need for a fast approximation of vulnerability. Mukherjee et al. introduced the *Architectural Vulnerability Factor (AVF)* [8] for measuring the reliability of microarchitectural structures. Unfortunately, AVF analysis requires a complex model of the underlying hardware, making it hard to be applied by software developers who usually do not possess the respective knowledge and tools. Sridharan et al. therefore introduced the *Program Vulnerability Factor (PVF)* [13], a software-based metric that does not require microarchitectural expert knowledge.

In this paper, we first describe a tool for computing the PVF for x86 applications (Section 2). We then use this tool in Section 3 to compute the PVF for the `nanojpeg` image decoder and compare the computed PVF to the results from a real FI campaign based on Fail* [11]. Our evaluation indicates that PVF analysis may serve as a starting point to answer the developer questions listed above. In Section 4 we then describe our ideas for addressing the few limitations PVF analysis still has.

2 Tool Support for Vulnerability Analysis

We implemented PVF/x86, a tool to perform PVF analysis on x86 binary programs.[3] The tool obtains an instruction trace for the binary in question and then computes the PVF based on this trace. To assess the quality of our computed PVF results, we used the Fail* fault injection framework to conduct a real-world FI campaign for comparison.

2.1 Computing the PVF of x86 Binaries

PVF/x86 provides two ways for generating an instruction trace: based on dynamic analysis or using an instruction pointer trace obtained from an external simulator. We perform dynamic analysis on Linux user-level applications using Linux' `ptrace` functionality. The easiest way to obtain a trace this way is to

[3] All source code is available at `https://github.com/TUD-OS/PVFAnalyzer`.

single-step the application and record all instruction pointer values on the way. However, this incurs a large execution overhead.

To reduce this overhead, we first perform a static analysis of the ELF binary in question and obtain the program's control flow graph (CFG). Based on the CFG, PVF/x86 determines the dynamic jump instructions in the binary. Thereafter, we run the application under `ptrace` control, but only instrument those dynamic jumps.

Once an instruction trace has been obtained, PVF/x86 performs PVF analysis by iterating over this trace. Such analysis requires a specific fault model to be imposed. In this paper we focus on analyzing bit flips in general-purpose registers of the CPU similar to Sridharan's work [13]. Iterating over the instruction trace, we generate a state sequence for each register. For each instruction, this sequence maps the state of every register to one of the states shown in Table 1.

`READ`	Instruction reads register
`WRITE`	Instruction writes register
`MODIFY`	Instruction reads&writes register
`IMPORTANT`	Register is not touched, but contains data important for program outcome
`DONTCARE`	Register is overwritten later, hence the current value does not influence program outcome
`UNKNOWN`	Register is not used, therefore no assertion can be made about its use

Table 1. Register access states for PVF analysis

State changes are detected by inspecting an instruction's operands using the `udis86` disassembler[4]. The disassembled information is unfortunately insufficient, because the x86 architecture includes instructions that *implicitly* access certain registers (e.g., `call` modifies the `ESP` register). Therefore, we added implicit constraints about such instructions to PVF/x86's parser component.

PVF/x86 finally computes a register's PVF from its state sequence. All instances where the register's state is in {`READ, WRITE, MODIFY, IMPORTANT`} are considered vulnerable. The register PVF for a given trace is then computed as the ratio of vulnerable instructions compared to the trace's instruction count.

2.2 FI Experiments with Fail*

The PVF metric (and our planned extensions) is intended to represent an approximation of a program's real (microarchitecture-agnostic) fault vulnerability. Therefore, the metric's quality can obviously be measured by analyzing the difference to real FI experiment results. We used Fail* [11], a versatile FI and experimentation framework, to inject faults into Bochs, a behavioral x86 simulator, and to observe the guest system's behavior afterwards. Fail* provides effective fault-space pruning techniques and can be configured to utilize large amounts of computing resources, which made it feasible to exhaustively visit the complete register-bit/instruction-offset space, allowing a one-to-one comparison with the PVF results.

[4] udis86 – disassembler library for x86 and x86-64. `http://udis86.sourceforge.net/`

3 Baseline evaluation: PVF vs. Fault-injection experiments

In the following we evaluate Sridharan's original PVF definition in the context of a case study involving a JPEG image decoder. Despite its example character, we believe the insights gained from a direct comparison between the PVF and results from a large-scale FI campaign will help in constructing an even more useful metric for application-specific program vulnerability.

3.1 Experiment setup

As a starting point for our comparative case study we chose `nanojpeg`[5], a JPEG decoder implemented in C – a typical code-base for a low-cost embedded system such as, e.g., a digital photo frame. An interesting property of `nanojpeg` is that any single FI run does not necessarily end with a binary *black-or-white* decision (target "crashed", vs. decoded JPEG is identical with the "golden run" output). Depending on the fault model (single-bit flips in the CPU's register file in our case) and what the developer application-specifically decides to be a "good" experiment outcome, a FI experiment may end up in any shade of gray: In the case of a JPEG decoder, the actual shade can be determined by calculating, e.g., the peak signal-to-noise ratio (PSNR) between the decoder's output and that of a fault-free run. The PSNR is a logarithmic measure approximating human perception of signal reconstruction quality, suitable for quantifying the image quality of a JPEG decoder run.

To keep the number of necessary FI experiments within a feasible magnitude, we selected a tiny (128×69 pixels, 2.58 kiB) JPEG input file[6] for our experiment runs. We compiled a bare-metal system image with gcc 4.4.5 (Debian 4.4.5-8) with full optimizations (`-O3`) enabled. The fault-free ("golden") run executes 3,729,437 instructions from entering its `main` function until the JPEG decoder finishes.

After applying conservative fault-space pruning techniques, we ran a total of about 170 million experiments with FAIL* to cover all possible register/bit/instruction-offset coordinates in our fault space, occupying our faculty's computing cluster for several days. Recorded experiment result details were, among others, successful program termination including the aforementioned PSNR metric, and several failure modes, including "crashes" like CPU exceptions (e.g., division by zero), accesses to memory regions outside the program's RODATA, DATA and BSS sections, or reaching a timeout after not triggering any of the other conditions for a reasonably long simulation time.

As outlined in Section 2, we feed the instruction trace from the golden run into our PVF/x86 tool. To stay in line with the results presented in [13], we calculate the PVF for trace blocks of 10,000 instructions each. For the `nanojpeg` trace, PVF/x86 ran less than 10 minutes on an off-the-shelf quad-core notebook.

[5] Version 1.2, `http://keyj.emphy.de/nanojpeg/`

[6] Felix Baumgartner, right before his jump from space in October 2012.

We also partition the FI results in chunks of 10,000 instructions, averaging the number of "good" experiment outcomes to facilitate a direct comparison.

3.2 A direct comparison of PVF and Fault-injection results

As mentioned before, the definition of "good" may vary depending on the application. In this subsection, any FI result diverging from the golden run output is classified as "bad", completely ignoring the PSNR quality metric or any other shades of gray for now.

Figure 1 shows the calculated PVF values right next to the actual results from the FI campaign, and accentuates the difference between both metrics in the bottom panel for each CPU register (negative difference values plotted in blue). At first glance it becomes clear that Sridharan's PVF is a pretty good, low-effort approximation for the time-consuming FI results we generated: The predictions for EAX (ignoring the first 100 blocks), EBX, ECX (aside the tail 50 blocks) and EDX (disregarding the first 50 blocks, and focussing on the trend instead of absolute values at around 250 blocks) are fairly accurate, making the FI resource spendings already look extremely wasteful. The results for EBP and the stack pointer, ESP, seem pretty off the mark instead, but we can offer an explanation for most anomalies in the diagram:

- In the first 100 instruction blocks, the decoder zeroes large memory areas, utilizing **EAX** for address calculations. In combination with a regularly applied bit masking operation, seemingly only half of the zero writes cause an abnormal experiment termination; as most simulator memory is zero at startup anyways, the lack of initialization does not disturb normal operations in these cases. Our current PVF implementation works on register (not bit) granularity, and ignores any data flow or bit-masking operations, explaining the 50% discrepancy here.
- The rise in **ECX** PVF–FI difference in the last 50 blocks is due to similar reasons: Again, a bit-masking operation comes into play, and additionally the probability to affect the final output image seems to continually decrease. The PVF, of course, ignores probabilities and is pessimistic in this case.
- The first 50 blocks in the **EDX** plot are simply explained by a deficiency of our implementation: Currently we isolatedly calculate the PVF for each block, ignoring all previous and following instructions. This issue blinds the analysis tool from seeing the address value living in EDX for 52 blocks before being used for actual memory accesses, being vulnerable to bit flips the whole time. A closer look reveals that the very same problem strikes the **EBP** and **ESP** PVFs, rising hopes that we can deal with this properly very soon.

4 Refining PVF Analysis

The initial results obtained in Section 3 indicate that the PVF is indeed an approximation for an application's susceptibility to hardware errors. By splitting

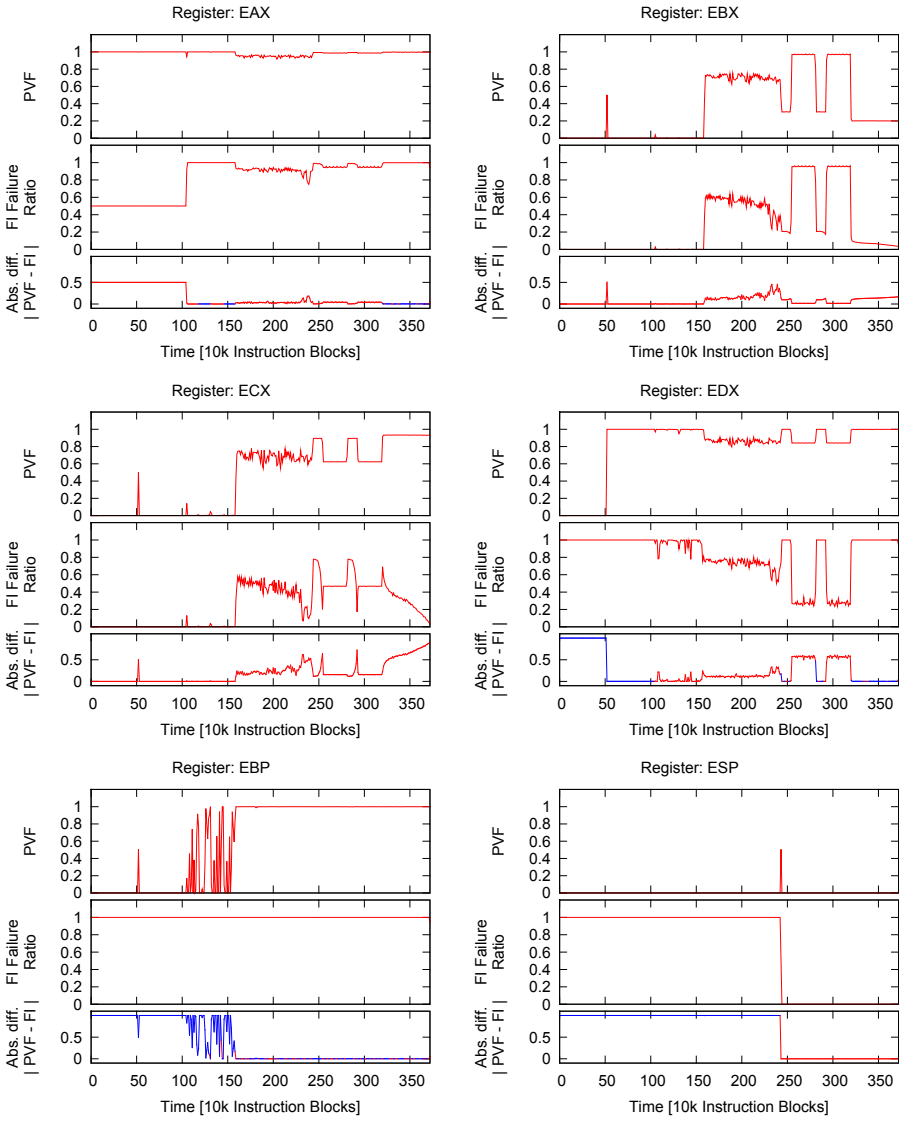

Fig. 1. Comparison of PVF and real program vulnerability for `nanojpeg-O3`. The first row of each subgraph shows the computed PVF for a specific register. The second row provides the results obtained using FI. The third row plots the absolute difference between the two (lower is better, negative values shown in blue).

application traces into relevant blocks (e.g., fixed length or at function boundaries), PVF analysis allows a software developer to pinpoint the most vulnerable parts of an application.

As already observed by Sridharan et al. [13], the PVF of two applications cannot directly be compared, because the absolute number of errors seen depends not only on the error probability (described by the PVF), but also on the runtime of the program. Therefore, to address the issue of comparing different application versions or algorithm implementations, the PVF needs to be combined with a runtime metric, such as the instruction count or wall-clock time.

4.1 Dealing with Quality Deviation

In order to more precisely assess the gray scale effect described in Section 3, information about the semantics of operations, i.e. *application knowledge*, is required. The usual approach to calculate the PVF assumes an error every time the decoder generates a picture different from the original one. However, our previous analyses of H.264 video decoding [5] show that for signal processing applications, certain classes of errors can be tolerated since they lead to a reduction in output quality, however they do not influence the program's control flow.

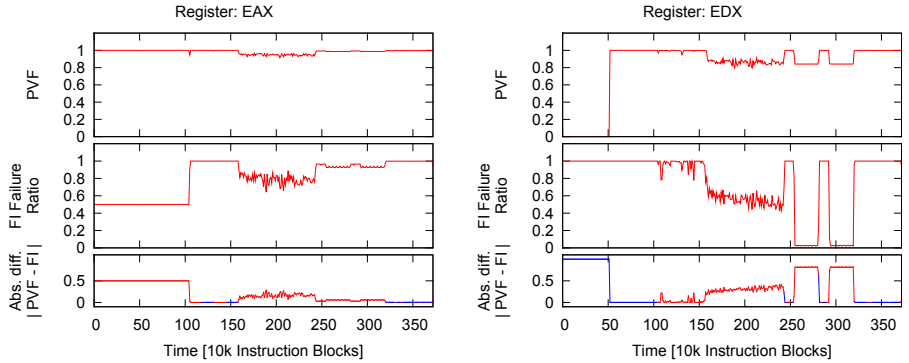

Fig. 2. Comparison of PVF and FI results when selecting a PSNR value of 20 as the quality threshold to decide whether a program run is considered an error.

To illustrate this effect, Figure 2 shows a comparison of PVF and FI, when classifying all images with a PSNR above 20 as correct outcomes. (A PSNR of 20 is considered a threshold for acceptable image quality.) Here, EAX shows a significant diversion between blocks 150 and 250, while EDX shows a higher absolute error than seen in Figure 1.

In order to identify such quality errors, we devised a classification [12] expressing the worst-case impact of an error on data objects with the help of newly introduced *reliable* and *unreliable* type qualifiers. We believe that in future work extending the compiler infrastructure to propagate this reliability information to registers, thereby bridging the semantic gap [3] between the source code and the machine code generated by the compiler, the results of PVF analyses will

give a more fine-grained assessment of error impacts. An analysis of the impact of compilers on the AVF [6] has already shown significant variations depending on the chosen level of optimization.

In addition, a compiler can also be used in order to improve the PVF of a program. A first investigation of the influence of different compiler backend optimizations on a program's PVF can be found in [13]. Here, the authors extend previous analyses by incorporating the tradeoff of a longer program runtime caused by optimizations reducing the PVF.

The availability of a fast PVF estimation tool such as PVF/x86 now allows to perform an online assessment of the PVF impact of different compiler optimizations on a program's PVF. By integrating a feedback loop, as previously investigated for energy and WCET optimizations [7], the compiler can perform a PVF-aware selection of applicable optimizations. First approaches in this direction were published by Rehman et al. [9]. However, their work concentrates on an optimization of instruction scheduling instead of an overall evaluation of optimization effects. Here, we expect a more general approach to provide improved results.

4.2 Incorporating Software Fault Tolerance Methods

One advantage of fault injection experiments over PVF analysis is that they easily allow evaluating the usefulness of fault tolerance methods, such as replication or operand encoding, with respect to fault rates. As the PVF is much faster to compute, our next question is: Can we incorporate software-implemented fault tolerance (SWIFI) methods into PVF analysis so that the PVF also allows estimating these mechanisms' effectiveness? The answer to this question depends on the respective SWIFT method.

Replication runs multiple independent instances of an application and compares the replicas' states at certain synchronization points [2]. Between such synchronization points, the replicas run identical code and are otherwise completely independent. N-way modular replication can tolerate $\frac{n}{2} - 1$ independent faults. Hence, an error will be masked unless there are $\frac{n}{2}$ or more errors within a single replication interval. The probability (PVF) for such an event is the product of the independent event probabilities (PVFs)[7], therefore:

$$PVF_{replicated}(reg) := PVF(reg)^{\frac{n}{2}} \tag{1}$$

The same formula also applies to mechanisms where the *compiler replicates data* across multiple registers [10]: Such errors will be masked by the compiler-generated code as long as the majority of values remains intact. For this specific interval in time, the registers contain the same data and therefore have identical PVFs. Hence, the probability of a failure is again the product of $\frac{n}{2}$ PVFs.

[7] Note, that the actual failure probability is the soft-error probability p multiplied with the PVF. We leave out p here, as it may be assumed to be constant for a given hardware platform.

Another interesting SWIFT method is the introduction of consistency checks either by the compiler or using (semi-)manual code modification [1]. The consistency checks make sure that at certain points in time errors will either be detected by the generated code or we can be sure that the registers contain correct data.

To analyze these methods, the PVF analyzer needs to be aware of them. The compiler may aid this analysis by providing information about the consistency checks, for instance by exporting the special instruction sequences used for checking. The PVF analyzer may then reset its analysis window whenever it encounters such a sequence.

To demonstrate how consistency checks influence the PVF, we performed another experiment based on `nanojpeg`. We assume that the compiler generates consistency checks either at certain instructions (for instance before a `ret` from a function) or periodically (e.g., every 100 instructions). We furthermore assume that the compiler somehow exposes this information so the PVF analysis can benefit from it.

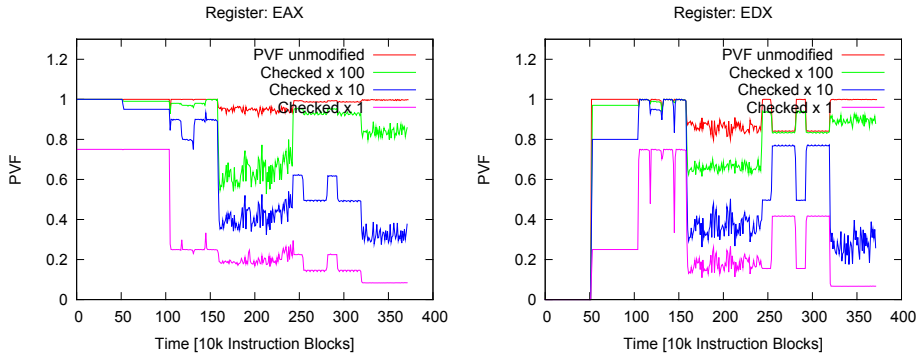

Fig. 3. PVF for different intervals of periodic register checking (100, 10, and 1 instructions respectively) compared to the originally computed PVF

Based on these assumptions we modified PVF/x86 to be aware of consistency checks and adapted the propagation of `IMPORTANT` register states by stopping this propagation at consistency checking boundaries. We show the PVF gains that can be expected by applying periodic consistency checking in Figure 3 (limiting ourselves to registers EAX and EDX for brevity). Note, that especially the graph for a checking interval of a single instruction is purely hypothetical, as the metric completely ignores the fact that the checking code itself will also be vulnerable to hardware errors.

5 Conclusion

In this paper we found that the PVF may serve as a starting point for estimating the vulnerability of x86 applications against hardware errors. Our experiments based on the `nanojpeg` JPEG decoder also pointed out that the PVF lacks

precision when dealing with applications that can generate data with varying quality. We believe that compiler assistance may help refining PVF analysis to this end. Finally, we demonstrated how the PVF may be used to analyze the impact of software fault tolerance methods on an application.

Acknowledgments

This work is supported by the German Research Foundation (DFG) priority program SPP 1500 under grants no. HA2461/8-1, SP968/5-1 and MA943/10-1.

References

1. Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Protecting the dynamic dispatch in C++ by dependability aspects. In *Proceedings of SOBRES*, LNI, pages 1–15. Springer, September 2012.
2. Björn Döbel, Hermann Härtig, and Michael Engel. Operating system support for redundant multithreading. In *Proceedings of EMSOFT*, October 2012.
3. Michael Engel and Peter Marwedel. Semantic gaps in software-based reliability. In *Proceedings of DFR'12*, Paris, France, January 2012. HiPEAC.
4. Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *Proceedings of ASPLOS*, pages 123–134, 2012.
5. Andreas Heinig, Michael Engel, Florian Schmoll, and Peter Marwedel. Improving transient memory fault resilience of an H.264 decoder. In *Proceedings of ESTIMedia*, Scottsdale, AZ, USA, October 2010. IEEE Computer Society Press.
6. Timothy Jones, Michael O'Boyle, and Oğuz Ergin. Evaluating the effects of compiler optimsations on avf. In *12th INTERACT Workshop at HPCA-14*, Feb 2008.
7. Paul Lokuciejewski and Peter Marwedel. Combining worst-case timing models, loop unrolling, and static loop analysis for WCET minimization. In *Proc. of ECRTS*, pages 35–44, Dublin / Ireland, July 2009.
8. Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of IEEE/ACM MICRO 36*, Washington, DC, USA, 2003. IEEE Computer Society.
9. Semeen Rehman, Muhammad Shafique, Florian Kriebel, and Jörg Henkel. Reliable software for unreliable hardware: embedded code generation aiming at reliability. In *Proceedings of CODES+ISSS*, pages 237–246, New York, NY, USA, 2011. ACM.
10. George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of CGO*, pages 243–254. IEEE Computer Society, 2005.
11. Horst Schirmeier, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, and Olaf Spinczyk. FAIL*: Towards a versatile fault-injection experiment framework. In *ARCS '12 Workshop Proc.*, volume 200 of *LNI*, pages 201–210. GI, March 2012.
12. Florian Schmoll, Andreas Heinig, Peter Marwedel, and Michael Engel. Improving the fault resilience of an H.264 decoder using static analysis methods. *ACM Transactions on Embedded Computing Systems*, 2012. (accepted for publication).
13. V. Sridharan and D.R. Kaeli. Eliminating microarchitectural dependency from architectural vulnerability. In *HPCA 2009. IEEE 15th International Symposium on High Performance Computer Architecture, 2009*, pages 117 –128, Feb. 2009.