How to protect the protector?

Tobias Stumpf
Chair of Operating Systems
TU Dresden - Germany
tstumpf@tudos.org

***Abstract*—This paper describes ongoing research to harden the lower part of the software stack, normally not covered by existing software-based fault-tolerance mechanisms. I discuss a combination of four techniques to harden the operating system kernel against hardware faults: resilient data structures, asynchronous checks, restartable OS services, and message protection. I present initial performance results for resilient data structures and message protection. The hardened operating system is the base layer for additional techniques like resilient I/O and resource management, a resilience-aware OS/application interface and exception handling, and usage of possible hardware extensions, which are briefly discussed.**

I. Introduction

This paper focuses on software-based fault-tolerance systems, which are based on COTS hardware. Figure 1 illustrates one possible setup. The left half of the figure is state-of-the-art. At the lowest level there is the operating system (OS) kernel. On top if it, there are OS services and fault-tolerance mechanism like replication [5] or checkpoint/restart (C/R) [4] to protect applications.

Application-level fault tolerance mechanism uses functionalities provided by lower layers. For instance, C/R implementations use operating system functionality to restart a crashed process and use storage or network functionalities to store and reload checkpoints. A replication manger (RM) needs at least a reliable voter. For resource management or to communicate with other applications the replication manager uses different OS services [5].

More generally, common application-level fault tolerance mechanism relies on the underlying software stack, which is known as reliable computing base (RCB) [6]. The RCB typically consists of hardware components as well as parts of the operating system and runtime layers. Because a fault within a replication manager or a C/R library affects the fault-tolerance of the protected application these functionalities also belong to the RCB.

In my work, I focus on the RCB. Even if an application level fault is more common, because of higher resource usage, a fault within the RCB is more critical. In the best case a bitflip only affects one application, but a fault at kernel level or inside any OS service compromises the whole system. While already presented fault-tolerance techniques rely on a RCB, different methods are necessary to protect the RCB.

Research challenges: This paper investigates software techniques to harden the RCB against transient and permanent hardware faults, to provide a fully protected software stack.

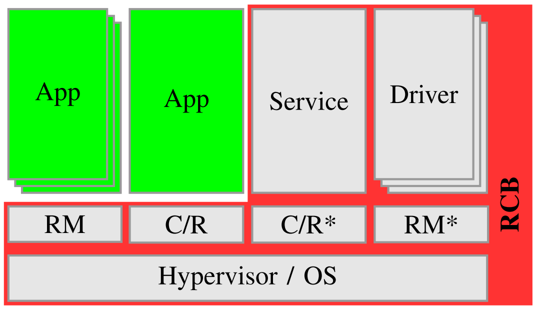

Fig. 1. Fault-tolerant system design. The green boxes are already hardened, but relys on the red part. An asterix indicates advanced techniques not yet available.

Section III starts with the protection of the lower layer. I analyzed an existing system (L4/Fiasco.OC) and developed a first prototype which includes resilient data structures and protected communication paths. The second part of this paper covers the whole RCB. The right half of Figure 1 illustrates an extended design, to protect further RCB components. Section IV describes methods to protect drivers as well as OS services.

II. Related Work

Compiler-based fault-tolerance techniques like SWIFT [7] or AN-Encoding [8] works independently from an underlying software stack. SWIFT works at the instruction-level and adds redundancy and control flow checks into the instruction stream, but works only for single-threaded applications. AN-Encoding uses arithmetic codes to detect errors. Because of the high encoding overhead, AN-Encoding is used to protect small but system critical components like a replication voter which runs on top of a fault-tolerant operating system [9].

Hoffmann et al. [9] compare techniques to harden a static and a small dynamic embedded real time OS against soft errors. In my work I will use a more powerful dynamic system which provides support for multiple address spaces. CuriOS [10], EROS [18] or Minix3 [11] are already used for fault-tolerant systems. CuriOS [10] provides an enhanced protection mechanism to minimize error-propagation between clients and provides C/R. EROS [18] enables system-level checkpoints and performs a whole system restart after a crash. Minix3 [11] is focused on software bugs inside device drivers. Otherworld [21] is designed to reduce the downtime after a crash occurs inside the Linux kernel. It installs a backup kernel during system boot, which is called after a system crash and uses the crashed system state as checkpoint.

Borchert et al. [13] presented an aspect oriented approach to protect OS data structures. An object can be protected by adding a checksum or replicating the class members. Before accessing any of the class members, a consistency check is performed to detect data corruption. Their approach is usable for most parts of the RCB, but fails for low-level code (like hardware initialization, OS entry/exit) usually written in assembly language.

III. Kernel Hardening

To reduce error propagation within the RCB my work relies on a microkernel design, in line with previous work ([10], [11], [18]). An OS kernel is the lowest layer and a failure in it affects the whole system. In a microkernel-based system OS service runs like an application. A fault within one system does normally not affect others. In contrast to previous work I am focusing on protecting the kernel itself. For evaluation I use L4/Fiasco.OC [12] a third generation microkernel. Nevertheless, the presented mechanism can be used to protect other RCB components like a replication manager or a C/R library as well.

To test the robustness of L4/Fiasco.OC and to identify starting points to increase its robustness we inject more than 200 million faults during the execution of L4/Fiasco.OC. Figure 2 illustrates a subset of the results for core functionalities of L4/Fiasco.OC. Around 50% of the bitflips are masked and have no effect on the execution. Corruption of kernel data structures often leads to a system crash, whereas a fault injected during communication leads to silent data corruption (SDC). Timeouts come from the test setup, because we stop the experiments after a certain point.

A. Resilient Data Structures

Because many crashes come from corrupted data structures (e.g. thread states, scheduling contexts or resource mappings), they should be encoded to detect and repair bitflips during runtime. To identify vulnerable data structure we repeated the fault-injection experience with assertions enabled. Around 30 % of the previous crashes triggered an assertion, indicating a corrupted kernel data structure. Even if the results from the assertion runs are not fully trustworthy, because some parts of the code include more fine-grained assertions than other code sections, they give a good indication which corrupted data structures leads to a system crash.

One fault-prone code section is the inter process communication (IPC) path. It is used for information exchange between applications as well as to call kernel functionalities. A lot of assertions are triggered within this path indicating that the state of the sender or receiver is broken. The state is represented with a bitmap, a typical data structure within an OS. A single bitflip changes the state and may also result in an invalid combination of set bits.

The first idea was to reuse the aspect oriented approach from Brochert [13], which was not possible, because existing aspect weavers [14] do not support state-of-the-art

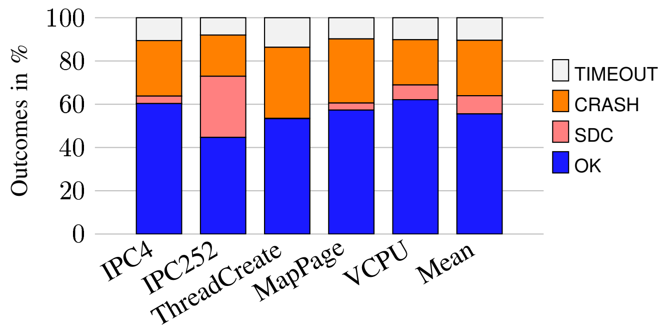

Fig. 2. Fault-injection experiments with L4/Fiasco.OC. Each bar illustrates one experimental setup and covers a basic microkernel functionality.

software techniques like templates in C++, which are used by L4/Fiasco.OC. To make my approach also reusable, I started to develop a library which provides resilient data structures.

The library already supports different bitmap implementations, including Hamming encoding, double modular redundancy with checksums (DMR) and triple modular redundancy (TMR). A first evaluation shows that TMR has the lowest performance impact, but the highest memory overhead. The execution overhead for Hamming encoding is more than one magnitude worser than TMR, because the encoding is fully implemented in software. The DMR implementation benefits from hardware support to calculate the checksums and therefore its performance overhead is closer to the TMR implementation.

In the next step I will add support for pointer-based data structures (like linked list or trees) which are common data structures within the RCB. I will reuse the work from Aumann and Bender [15] and Jorgensen et al. [16], who analyzed resilient linked list and priority queues.

B. Asynchronous Checks

Memory scrubbers scan the memory periodically, to detect memory corruption at an early state, to minimize error-propagation. Memory scrubbers can also reduce overhead if a fault is detected and repaired before the memory is read within the normal execution path. Hwang et al. [17] identified drawbacks of existing hardware-implement memory scrubbers. Existing memory scrubbers have a low impact on detecting faults, because ECC can only repair single bitflips and the scan rate of around one GB per hour is too low.

I will take these drawbacks into account and develop a software-based memory scrubber, which is based on two different methods. The first one uses the previously described data structures by validating the checksum or comparing redundantly stored data. The second one is more complex and uses OS-specific knowledge. For instance, a thread cannot be in the ready list while waiting for an external event. With a higher scan-rate and superior detection and correction opportunities, I assume to overcome the disadvantages of hardware-implement memory scrubbers.

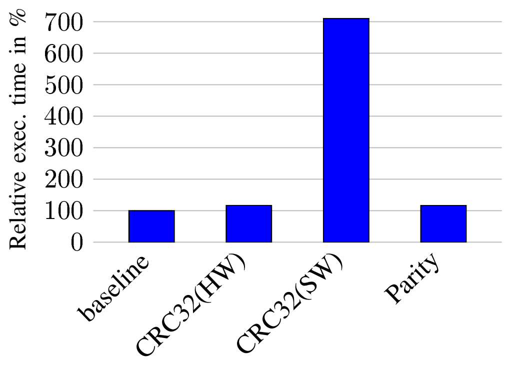

Fig. 3. Runtime overhead for different IPC protection mechanisms.

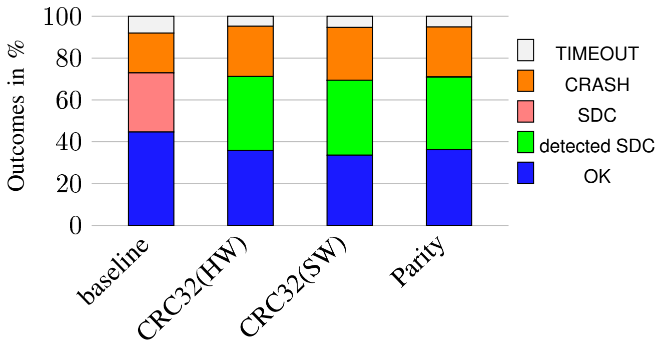

Fig. 4. Repeated IPC252 fault-inject experiments from Figure 2 with different protection strategies.

C. *Protecting Messages*

The fault injection experiments from Figure 2 show varying silent data corruption for each experiment. A detailed evaluation showed that most of the SDC comes from the payload of IPC messages. Because the number of processors and the corresponding interconnection network increases, I assume an increased fault rate during communication. To detect corrupted messages I use checksums. I include the network encoding functionality inside the system's software to make it transparent for the application avoiding expensive rewrites. Figure 3 shows the overhead of encoded IPC messages and Figure 4 the influences on the SDC.

D. *Restartable services*

A fault within one OS service can crash the whole system, because a mandatory functionality (e.g. memory management) is not available anymore. A corrupted service normally does not influence other service, because a microkernel isolates the different components. But, without any recovery mechanism for a single service a whole system restart is necessary.

In my work, I will reuse existing C/R approach like CuriOS to restart services. In contrast to CuriOS I will enhance the C/R functionality to enable restartable OS services. Reloading a server often requires a reset of its clients to ensure a consistent system state. For core components like memory management, this means resetting all clients. To avoid a global reset, I will use a logging mechanism to avoid information loss between the last checkpoint and the crash of the service.

IV. Reliable System Design (Outlook)

Operating systems traditionally optimize for speed when managing resources and mitigating I/O accesses. In order to improve fault tolerance I aim to make reliability another optimization criterion by replicating device drivers and I/O and making resource management aware of the underlying hardware platform's dependability properties.

A. *Replicated Device Drivers*

Existing device driver specific techniques mainly consider software failures [11], [19], [20]. Developing a hardware-fault resilient driver, raises additional challenges compared to application level fault tolerance. For instance, many read and write operations cannot be repeated, because they affect other systems or have a different result.

I will use replication to overcome the problem that a broken driver performs faulty device access. The replicated driver works like a replicated application. The replication manager compares I/O accesses and performs or denies them. Therefore, a fail safe design for the replication manager is necessary. Because a fault can occur at any time the vulnerability window between checking the replicated state and performing the hardware access should be minimized.

B. *Resilient Resource Allocation*

A fault at a specific memory address increases the fault probability for related memory cells [17]. Hence, the resource allocation strategy affects the reliability of higher level fault tolerance techniques. In my work I will consider the results from Hwang et al. [17] to create a resilient allocator taking the interferences between different memory areas into account.

C. *OS / Application Interface*

Software fault tolerance mechanisms are mainly per-application. The OS has probably more knowledge about broken hardware, but has no application knowledge. Therefore, I will rework the OS interface with reliability in mind. For instance, after detecting a memory fault, the OS should notify all applications which have allocated probably affected memory. An application specific fault handler has more knowledge to detect and repair a fault (e.g. checking checksum, recalculating or reading the data from the hard disk).

The OS can use hardware mechanisms to detect corrupted memory. If an application uses checksums or replication, it can also detect faults. To use this information, the OS needs an interface to communicate this knowledge from the application to the OS.

D. *Resilient-aware Exception Handling*

Exceptions are often interpreted as bugs and the application is terminated. Taking hardware faults like bitflips into account, we should change this behavior. For instance, if a page fault occurs and the corresponding memory is not allocated by the application, this may indicate a bitflip. An application can provide a handler which is called after the page fault to check the memory address.

Data corruption within the kernel can result in a kernel panic, stopping or rebooting the system. Like Otherworld, I will avoid a system reboot, but I will trigger a consistency check to bring the kernel into a consistent state and continue execution.

E. Hardware Extensions

Bitflips in hardware based data structures like page tables, or interrupt vector tables, can lead to non-recoverable failures. To detect and repair errors in this data structures a changed hardware design is necessary. Nevertheless, the software can perform periodic consistence checks to detect faults before the data is used.

I will further investigate how I can use existing mechanisms like Intel machine check exception to react on a hardware fault before a crash occurs. Other techniques like instruction pipeline fingerprinting [22] can be used to check for correct instruction decoding and handle a pipeline error within the RCB.

V. Conclusion

Current fault tolerance approaches often protect only the application code and rely on an RCB. Within my PhD I will investigate software methods to protect the RCB. The result will be a software stack that provides detection and recovery from hardware faults at all levels.

Acknowledgment

I would like to thank Björn Döbel for his extensive fault-injection experiments, which I used as starting point to evaluate error-proneness of L4/Fiasco.OC and Maurice Bailleu, who implemented the different IPC protection mechanisms.

This work was supported by the German Research Foundation (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500 - http://spp1500.itec.kit.edu) and within the Cluster of Excellence 'Center for Advancing Electronics Dresden' (cfAED).

References

[1] R. Baumann, "Soft Errors in Advanced Computer Systems," *IEEE Des. Test*, vol. 22, pp. 258–266, May 2005.

[2] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM Errors in the Wild: A Large-scale Field Study," in *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, (New York, NY, USA), pp. 193–204, ACM, 2009.

[3] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, a. gara, G. Chiu, P. Boyle, N. Chist, and C. Kim, "The ibm blue gene/q compute chip," *IEEE Micro*, vol. 32, pp. 48–60, Mar. 2012.

[4] J. S. Plank, "An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance," Tech. Rep. CS-97-372, University of Tennessee, July 1997.

[5] B. Döbel, H. Härtig, and M. Engel, "Operating system support for redundant multithreading," in *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, (New York, NY, USA), pp. 83–92, ACM, 2012.

[6] M. Engel and B. Döbel, "The reliable computing base - a paradigm for software-based reliability," in *GI-Jahrestagung*, pp. 480–493, 2012.

[7] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, (Washington, DC, USA), pp. 243–254, IEEE Computer Society, 2005.

[8] U. Schiffel, M. Süßkraut, and C. Fetzer, "An-encoding compiler: Building safety-critical systems with commodity hardware," in *SAFECOMP '09: Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security*, (Berlin, Heidelberg), pp. 283–296, Springer-Verlag, 2009.

[9] M. Hoffmann, C. Borchert, C. Dietrich, H. Schirmeier, R. Kapitza, O. Spinczyk, and D. Lohmann, "Effectiveness of fault detection mechanisms in static and dynamic operating system designs," in *Proceedings of the 17th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '14)*, pp. 230–237, IEEE Computer Society Press, June 2014.

[10] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "Curios: improving reliability through operating system structure," in *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation*, OSDI'08, pp. 59–72, USENIX Association, 2008.

[11] B. G. P. H. Jorrit N. Herder, Herbert Bos and A. S. Tanenbaum, "Construction of a highly dependable operating system," in *Proc. of EDCC'06*, (Coimbra. Portugal), October 2006.

[12] A. Lackorzynski and A. Warg, "Taming subsystems: capabilities as universal resource access control in L4," in *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, IIES '09, pp. 25–30, ACM, 2009.

[13] C. Borchert, H. Schirmeier, and O. Spinczyk, "Generative software-based memory error detection and correction for operating system data structures," in *Proceedings of the 43nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*, IEEE Computer Society Press, June 2013.

[14] aspectc.org, "A set of c++ language extensions to facilitate aspect-oriented programming with c/c++."

[15] Y. Aumann and M. A. Bender, "Fault tolerant data structures," in *37th Annual Symposium on Foundations of Computer Science, FOCS '96, Burlington, Vermont, USA, 14-16 October, 1996*, pp. 580–589, IEEE Computer Society, 1996.

[16] A. G. Jørgensen, G. Moruz, and T. Mølhave, "Priority queues resilient to memory faults," in *Algorithms and Data Structures, 10th International Workshop, WADS 2007, Halifax, Canada, August 15-17, 2007, Proceedings* (F. K. H. A. Dehne, J. Sack, and N. Zeh, eds.), vol. 4619 of *Lecture Notes in Computer Science*, pp. 127–138, Springer, 2007.

[17] A. Hwang, I. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: Understanding the nature of dram errors and the implications for system design," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (New York, NY, USA), pp. 111–122, ACM, 2012.

[18] J. S. Shapiro, J. M. Smith, and D. J. Farber, "EROS: a fast capability system," in *Proceedings of the seventeenth ACM Smposium on Operating Systems Principles*, SOSP '99, pp. 170–185, ACM, 1999.

[19] A. Kadav, M. Renzelmann, and M. M. Swift, "Fine-grained fault tolerance using device checkpoints," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (Houston, TX), March 16-20 2013.

[20] M. M. Swift, S. Martin, H. M. Levy, and S. J. Eggers, "Nooks: An Architecture for Reliable Device Drivers," in *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, (New York, NY, USA), pp. 102–107, ACM, 2002.

[21] A. Depoutovitch and M. Stumm, "Otherworld: giving applications a chance to survive OS kernel crashes," in *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, (New York, NY, USA), pp. 181–194, ACM, 2010.

[22] P. Axer, R. Ernst, B. Döbel, and H. Härtig, "Designing an analyzable and resilient embedded operating system," in *Proc. on Software-Based Methods for Robust Embedded Systems*, (Braunschweig, Germany), 2012.