Operating System Services for Recovering Errant Applications

Raul Barbosa
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
rbarbosa@ce.chalmers.se

Abstract

Operating systems often manage critical infrastructures where failures can have serious consequences. This raises great concerns about their robustness. From the user perspective, it is the service delivered by host applications that needs to be dependable. Operating systems should therefore provide comprehensive error detection and recovery services to those applications, so that the system as a whole can be dependable and secure. This paper addresses the recovery flow that takes place after an application error is detected. The goal is to combine existing techniques into a set of operating system services that support application recovery both from software and hardware errors. We describe a prototype system where these services are currently being implemented and outline how we intend to experimentally evaluate them.

1 Introduction

Designing robust operating systems has been the goal of numerous research endeavours since the late 1950s, when the first mainframe computers were born. Since then, operating systems have managed critical infrastructures ranging from server rooms to embedded devices, as well as crucial user information on desktop computers. Given that a failure of such computers can have serious consequences, the operating systems must be reliable in the presence of faults. Moreover, they should also provide comprehensive error detection and recovery services to hosted applications, so that the system as a whole can be dependable and secure.

Computer systems are affected by faults of diverse origins. Software defects – introduced during development – can cause errors in the applications and in the operating system itself. For this reason, a large body of research has been devoted to preventing and removing software defects; this is, however, a complex problem lacking perfect solutions. Thus, operating systems have to cope with the existence and occasional activation of software faults, *e.g.*, when a buggy line of device driver code is executed. Another important source of errors are hardware faults – mostly of physical nature – due to environmental factors, aging, etc. These are now more concerning than ever, largely due to modern manufacturing processes and the associated variability and degradation of microprocessors [3, 13, 17].

This paper is chiefly concerned with detecting and recovering errant applications. Ultimately, from the user perspective, it is the service delivered by the applications that needs to be dependable. Recovering applications (in addition to isolating them) is therefore key to ensuring sustainable delivery of service. Accordingly, we should judge the dependability of an operating system not only for its resilience to failures but also for the services it provides to hosted applications with regards to error detection and recovery.

To this end, operating systems can make use of a vast multitude of techniques available in the literature. In this paper we address only those targeting hardware and software faults of non-malicious nature. Our goal is to combine several such techniques into a set of operating system services that detect application errors and support their recovery.

The first obstacle is that the recovery flow usually depends on whether the error was caused by hardware or software. This is generally hard to diagnose. Consider an example where the memory management hardware raises an exception indicating an erroneous memory access; it may have been caused by a missing pointer initialization in the software but also by corruption of a pointer due to a hardware fault. In this case it is difficult to choose, for instance, whether to rollback and retry the operation or to transfer control to a user-mode exception handler.

Let us assume that application errors can be detected by the operating system even though their cause is unknown. Our proposal is to consider at first that an error was caused by a transient hardware fault. To cope with these, we checkpoint applications frequently and rollback to a previous state upon error detection. If the cause was indeed a hard-

ware fault, there's a high chance that it will vanish. Otherwise, if an error is detected again before another checkpoint is made, we must assume that it was caused by a software fault. At this point we transfer control to an application-specific exception handler where the designer can decide what should be done. If no such handler is defined, the operating system may simply restart the application – this is often sufficient to deal with rarely activated bugs.

We are developing a prototype system which we intend to use for experimentally assessing various techniques aimed at building robust operating systems. The remainder of this paper details the design principles of the prototype and outlines the experimental evaluation. We begin by surveying related research in Section 2. Error detection is discussed in Section 3 and Section 4 elaborates on the strategy for recovering applications. The experimental evaluation through fault injection is discussed in Section 5. Lastly, we summarize the main conclusions of the paper in Section 6.

2 Related Research

A great deal of work has been dedicated to ensuring that operating systems are resilient to internal failures. In this context, kernel extensions such as device drivers are usually identified as the major source of problems. The *microkernel* approach attempts to solve this issue elegantly by isolating kernel extensions in user-mode, where fault containment can be more easily achieved. Herder *et al.* [6] use this design principle in the Minix operating system. There is a price to pay for the increased reliability: obtaining an operating system service often involves full context switching and additional data copying. Tanenbaum *et al.* [15] point out that this performance penalty is worth the trade-off in systems where reliability is the main concern.

Swift *et al.* [14] use a different strategy to tackle the same problem. Their approach, implemented in Nooks, uses the more common *monolithic* kernel structure where extensions run in kernel-mode. It should be emphasized that kernel-mode instructions access main memory through the memory management hardware – just like user-mode instructions. The difference is that user-mode execution has restricted access to privileged registers and instructions. If we abstract from malicious faults, device drivers can be isolated by marking unnecessary pages as read-only during their execution. The authors make use of this feature to implement *lightweight protection domains*. Additionally, they propose the usage of wrappers to monitor control-flow between the drivers and the kernel.

A different approach is taken by Hunt *et al.* in designing the Singularity operating system [7, 8]. The core of Singularity is a microkernel written almost entirely in Sing# – a type-safe programming language with low-level constructs. Instead of relying on isolation through hardware protection, device drivers and applications are executed in the context of *software-isolated processes*. Here, the fundamental principle is using static software verification to assure that untrusted components are unable to access memory outside their context. Moreover, interaction among programs takes place via contract-based channels that can be statically checked; and the programs themselves come with a manifest allowing static and dynamic verification of key properties. The emphasis is thus on fault prevention rather than fault tolerance.

When it comes to detecting application errors, operating systems make use of techniques such as memory protection and watchdog timers to ensure spatial and temporal isolation. Under these circumstances, faults may still propagate via the system call mechanism – the door used by applications to request operating system services and to reach the device drivers. Peterson *et al.* [11], Provos [12] and many others propose the usage of *sandboxing* as the means to protect the system call mechanism. This technique consists of interposing the access to system calls with a filter that enforces a given policy. In practice, this serves as an accurate error detection mechanism.

After an error is detected, the recovery process is often left to the application designer or the operating system simply terminates the application. As we described before, we adopt the well-known checkpoint and rollback method to recover applications from transient hardware failures. This method can also be applied directly by the hardware. Wang and Patel [16] use microarchitecture-level checkpointing and define a set of error symptoms which trigger the processor to return to a previous state. Since we assume a generic hardware platform, we opt for implementing checkpointing as an operating system service.

The effectiveness of a rollback recovery depends substantially on the checkpointing strategy. If checkpoints are too frequent, their overhead is too high; if they are too sparse in time, recoveries may require too much recomputation. The work by Ling *et al.* [10], among others, addresses the checkpointing strategy problem. They offer a mathematical model for optimal checkpointing under time-varying failure rates. Results of this kind are an important input to our design, as they provide the means to choose the optimal checkpointing strategy.

3 SECERN: An Extension to MicroC/OS-II

A current trend in the design of embedded systems is to allow multiple applications to share a common hardware platform. The goal is to avoid using dedicated microcontrollers in order to achieve scalability as the number of functions grows. Initiatives such as the standard interface for avionics applications, defined by the ARINC 653 specification [2], aim at developing the critical infrastructures

and operating systems that support this level of integration. Since these initiatives target safety-critical systems, a fundamental concern is to ensure that resource sharing can be accomplished in a safe and reliable manner.

We are developing a prototype system intended for experimentally assessing various techniques for building robust operating systems. The prototype is based on the MicroC/OS-II real-time kernel, designed by Labrosse [9]. The kernel's source code is very well documented and freely available for academic purposes, making it amenable to extension.

MicroC/OS-II lacks support for isolating applications from one another and from the operating system. To solve this, we began by extending the kernel with memory protection [1] using a Memory Management Unit (MMU). This was accomplished by making the distinction between processes and threads, where each process owns a private address space that contains one or more threads of execution. Memory allocation is done statically by the linker – a common design decision in embedded systems.

A distinguishing feature of our implementation is that the Translation Lookaside Buffer (TLB) – a small cache that speeds up address translation – is updated during context switch. Our approach is to update the TLB with the pages that belong to an application before running that application. Thus, we avoid handling TLB-miss interrupts during the normal execution. This, in turn, makes it simpler to determine the worst-case response time of programs (a very important issue in real-time systems).

The extended version of the operating system runs on a computer board featuring a Freescale MPC5554 microprocessor based on the PowerPC architecture. On this processor, the time needed for a full context switch without updating any TLB entries is 10 μs. Considering a typical embedded application with 4 pages of memory, it takes 31 μs for context switching; with 8 pages it takes 53 μs. However, this increased time is not a penalty, as handling TLB misses would be more expensive in the worst case and less deterministic.

3.1 Error Detection in SECERN

In addition to memory protection, our goal is to further develop the operating system extension – denominated SECERN – by combining several error detection and recovery techniques. Some of those techniques are based on the literature surveyed in Section 2, adapted to suit the performance needs of embedded applications. Below, we describe the error detection mechanisms that are implemented in the current version of SECERN.

Memory protection. Each process has a private memory space statically allocated by the linker. Accessing memory outside this address space raises a CPU exception that is handled by SECERN.

Processor exceptions. All other CPU exceptions are handled by SECERN as well. Many are classified as *recoverable*, meaning that they were caused by an application error and the remaining parts of the system are intact. In such cases, application recovery is triggered.

Unauthorized system calls. Our extension supplies the ID of the calling process through a kernel structure that can be used to filter erroneous accesses to the system call. The ID can be checked by the system call's exception handler in order to enforce a given access policy.

Invalid system call parameters. This mechanism uses the caller ID principle targeting the device drivers. Any kernel extension can make sanity checks on the parameters passed by the caller. An example may be a communication driver that checks the size of a buffer and reports an error of the caller application.

Task deadline missed. In SECERN, a thread is expected to release the processor before its deadline. A scenario where this does not happen is considered an error. We note that not all tasks are required to have a deadline (*e.g.*, background tasks). These should execute with low priority to ensure that they cannot interfere with other programs.

Application-specific checks. The application designer is able to report errors to the operating system. In this case, the operating system will attempt rollback recovery as if the error had been detected by any of the previous mechanisms.

4 Recovering Errant Applications

The main contribution of this paper is the recovery flow that takes place after an application error is detected by one of the above-described mechanisms. The goal is to provide a comprehensive method that allows applications to be recovered both from hardware and software errors. To make this possible, one must diagnose the cause of each error, so that the appropriate recovery technique can be invoked.

Our proposal is to checkpoint the applications frequently and attempt rollback recovery upon error detection. There is a high chance that the error will vanish if it was caused by a transient hardware fault. However, it may also be the case that an error is detected again (before the next checkpoint). In this case, we diagnose it as being caused by software. The operating system can be configured to choose, at that point, one of two options: (i) restart the task; (ii) transfer control to an application-specific exception handler. In the

second case, the designer may decide, for example, to implement a backup routine that produces a degraded result which is known to be safe.

One must also consider the case when an error is detected a third time, *i.e.*, when both recovery attempts fail. Such cases may be caused by permanent hardware faults which prevent an application from executing correctly. Under such circumstances, the application is terminated in order to ensure fail-silence and to constrain resource consumption.

4.1 Checkpointing Strategy

The checkpointing strategy was identified earlier in the paper as determinant for the performance of the recovery process. Essentially, we need to know *what* part of the state should be saved and *when* it should be saved. Regarding what to save, we intend to begin by implementing an agnostic strategy, where the entire address space and processor registers in the context of an application are saved. There are more efficient approaches but our first goal is to evaluate the effectiveness of our proposed method, rather than obtaining the best performance.

Regarding the moment when checkpoints should be made, we propose to have a user-mode thread, with permission to read everywhere, copying the state of applications when they are idle. This way we can make use of the available slack time to increase the dependability of the system. To achieve this, the operating system should provide a service which applications use to request checkpoints. Those requests are queued and serviced by the low-priority checkpointing thread.

To ensure a checkpoint's consistency, we must guarantee that it is completed within one idle period of the application being checkpointed. The concern here is that the checkpointing thread can be preempted by an application while saving its state. When this occurs, the checkpointing thread must restart copying the application's state, since it may have been modified during the execution of the application. For this reason, the context switching routine should notify the checkpointing thread whenever an application is executed. This can be done by updating a single value in an array containing one element for each application, thereby keeping the overhead minimal.

4.2 An Alternative to Checkpointing

There are other techniques for masking hardware errors that can be used instead of checkpoint and rollback. We consider using *time redundancy* by executing the same task a second time when an error is detected. This method is appropriate if we consider the classic task models from which the *rate monotonic* and *earliest deadline first* results were derived. Here, tasks are issued periodically with an input and must produce their output by a given deadline. As long as task deadlines are met, we can use time-redundant execution to recover from transient failures.

From a practical viewpoint, running the task a second time upon error detection is equivalent to restoring a pseudo-checkpoint at the beginning of the task's execution. The advantage is that it does not require saving the state of applications, which may be a time-consuming activity, depending on the implementation. This method is appropriate for *stateless* tasks where each iteration is only dependent on its input. Moreover, we can also use this technique for tasks that depend on an external state that they cannot update (*e.g.*, the current operation mode); in this case we can view the external state as part of the input to the task.

5 Evaluation Outline

In this section we outline an experimental evaluation of the techniques described throughout the paper. Our goal is to assess, using fault injection, the robustness of the SECERN extension with regards to error detection and recovery. First, we are interested in ensuring that faults in an application do not propagate to the operating system nor to other applications (*i.e.*, isolation). Second, the proposed recovery flow must be validated and evaluated to ensure its usefulness and practicality.

With this in mind, we have developed a fault injection tool, for our experimental setup, that can inject faults and monitor the outcomes. The evaluation should use two independent processes receiving input and producing output using simple communication primitives. The tool injects faults in one of the applications and intercepts the results (of both applications) in the form of value/time pairs. The results are then classified regarding their correctness/timeliness and the activation of error detection mechanisms is monitored.

We consider injecting both hardware and software faults in our experiments. Hardware faults are emulated through bit-flips in registers and memory. Regarding software faults, we intend to use the most representative operators found by Durães and Madeira in their field study of open-source programs [4]. It should be noted that we must inject software faults manually in the source code of applications, as we have no automatic tool (*e.g.*, [5]) for this purpose. Nevertheless, we still depend on the fault injection tool for monitoring and classifying the outcome of each fault injection experiment.

5.1 Research Questions

We have identified a number of research questions which we expect to address during the experimental evaluation. The most important ones are:

How well does SECERN handle faults? In general, we're interested in understanding how robust operating systems can be built (focusing on embedded applications). To this end, fault injection is a good approach to testing, for instance, the coverage of the error detection mechanisms. A possible outcome would be to identify vulnerabilities in our extension.

How effective is each error detection mechanism? Each technique has a certain probability of detecting hardware and software errors. When several techniques are combined, there is a chance that some of them overlap and that a given technique is not effective. Answering this question allows us to make an informed decision on which set of mechanisms is the best.

Does memory protection ensure isolation? One can assume that the hardware itself has no design faults. However, there may be hidden faults in the tightly coupled software that manages that hardware. The goal here is to evaluate the isolation of applications upon hardware and software failures.

What is the impact of hardware faults? It is known that a hardware fault can potentially disrupt an entire hardware unit. However, the operating system only executes 5% of the time for a typical workload [9]. During the remaining time the processor is either idle or running applications. It is therefore likely that many transient hardware faults will only affect the context of a single application, much like software faults. We intend to estimate how likely this is.

Is the recovery flow effective and efficient? In this paper we proposed a recovery flow that addresses both hardware and software faults. Using fault injection we can evaluate the benefits of using this approach, the overhead it imposes and the latency of the recovery process.

6 Conclusion

This paper addresses the problem of recovering errant applications. We argue that an operating system's dependability should be judged not only for its resilience to failures but also for the services it provides to hosted applications with regards to error detection and recovery. The reason for this is that the applications are responsible for providing service to the system users. Thus, isolating faulty applications from one another (and from the operating system) is fundamental, but recovering them is also essential to ensure sustainable delivery of service.

Effective error detection is the premise to a successful recovery. We describe the error detection mechanisms implemented in SECERN – our extension to MicroC/OS-II which aims at providing robust fault containment around software components. This extension provides, among other mechanisms, hardware-supported memory protection. This feature is intended to prevent faults in one component from propagating to other components through the memory spaces. Memory protection is uncommon in embedded systems due to, among other factors, its impact on the worst-case response time of applications. However, we have opted for inserting, in the TLB, the page table entries of a process while context switching to that process. In this way, we avoid handling TLB-miss interrupts during the execution of applications, thereby eliminating the unpredictability of memory accesses.

In this paper we propose a recovery flow that takes place after an application error is detected. The method consists in assuming, at first, that an error is caused by a hardware fault and attempting rollback recovery. If the error is detected again (*i.e.*, the rollback recovery is unsuccessful), its cause is diagnosed to be a software fault. In this case, the principle is to allow the application designer to choose from a set of options. One option is to transfer the control to an application-specific exception handler where a different version of the software is executed; another option is to simply restart the application. Indeed, one of the principles adopted in this paper is to provide the application designer with many different options for recovery, so that the best choice can be made depending on the needs of each system.

Lastly, we outline an experimental evaluation of the proposed operating system extension. The goal is to assess, using fault injection, various techniques aimed at building robust operating systems. To clarify the desired outcomes of this evaluation, we identify a number of important research questions. We intend to complete the implementation of the recovery flow and address those research questions in the near future.

References

[1] J. Alçada, R. Barbosa, and J. Karlsson. Memory protection in a real-time kernel (fast abstract). In *Proceedings of the 6th European Dependable Computing Conference (EDCC-6), Supplemental Volume, Coimbra, Portugal*, Oct. 2006.

[2] ARINC Incorporated. ARINC specification 653-1: Avionics application software standard interface, Oct. 2003.

[3] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, Nov.-Dec. 2005.

[4] J. Durães and H. Madeira. Definition of software fault emulation operators: A field data study. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003), San Francisco, CA, USA*, pages 105–114, June 2003.

[5] J. Durães and H. Madeira. Emulation of software faults: A field data study and a practical approach. *IEEE Transactions on Software Engineering*, 32(11):849–867, Nov. 2006.

[6] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Failure resilience for device drivers. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2007), Edinburgh, UK*, pages 41–50, June 2007.

[7] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, Apr. 2007.

[8] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. D. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, Oct. 2005.

[9] J. J. Labrosse. *MicroC/OS-II: The Real-Time Kernel*. CMP Books, second edition, 2002.

[10] Y. Ling, J. Mi, and X. Lin. A variational calculus approach to optimal checkpoint placement. *IEEE Transactions on Computers*, 50(7):699–708, July 2001.

[11] D. S. Peterson, M. Bishop, and R. Pandey. A flexible containment mechanism for executing untrusted code. In *Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, USA*, pages 207–225, Aug. 2002.

[12] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium, Washington, DC, USA*, pages 257–272, Aug. 2003.

[13] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The impact of technology scaling on lifetime reliability. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN 2004), Florence, Italy*, pages 177–186, June-July 2004.

[14] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1):77–110, Feb. 2005.

[15] A. S. Tanenbaum, J. N. Herder, and H. Bos. Can we make operating systems reliable and secure? *IEEE Computer*, 39(5):44–51, May 2006.

[16] N. J. Wang and S. J. Patel. ReStore: Symptom-based soft error detection in microprocessors. *IEEE Transactions on Dependable and Secure Computing*, 3(3):188–201, July-Sept. 2006.

[17] M. Zhang, T. M. Mak, J. Tschanz, K. S. Kim, N. Seifert, and D. Lu. Design for resilience to soft errors and variations. In *Proceedings of the 13th IEEE International On-Line Testing Symposium (IOLTS 2007), Crete, Greece*, pages 23–28, July 2007.