Evaluation of Software-Based Fault-Tolerant Techniques on Embedded OS's Components

Hosein Mohammadi Makrani[1], Amir Mahdi Hosseini Monazzah[2], Hamed Farbeh[3], and Seyed Ghassem Miremadi[4]
Department of Computer Engineering
Sharif University of Technology
Tehran, Iran 1155-9517
Email: [1]makrani@ce.sharif.ir, [2]ahosseini@ce.sharif.edu, [3]farbeh@mehr.sharif.edu, and [4]miremadi@sharif.edu

***Abstract*—Software-based fault-tolerant techniques at the operating system level are an effective way to enhance the reliability of safety-critical embedded applications. This paper provides an analysis and comparison of five well-known recovery techniques, i.e., micro rebooting, recovery block, N-Version Programming (NVP), micro extension, and transactional extension for an embedded operating system's components, from performance point of view. These techniques are applied without any modification on the main architecture of the operating system. The techniques are implemented on a virtual ARM Integrator board which is emulated by the QEMU software (2.0.0) under the control of Embedded Linux operating system (3.9.0). The totals of 5000 software errors are ignited using a simulation environment. The results show that the recovery time overhead varies between 0.17% and 0.67%, and the performance overhead varies between 5.81% and 218.65% depending on the techniques.**

Keywords-embedded operating system; fault tolerant; recovery; performance.

I. INTRODUCTION

Nowadays, the embedded systems are employed as crucial control components in safety-critical and real-time areas such as medical devices, automobile, and aviation. To maintain the dependability of such applications, several fault tolerance techniques have been proposed in the recent decades.

In the recent years, the improvements in the performance of hardware devices have led to excessive attentions to software fault tolerance techniques. The software fault tolerance techniques can be implemented at the application code or operating system of an embedded system. Applying the fault tolerance techniques in an operating system allow the designers to develop their application without worrying about the dependability of the whole system. Hence, operating system approaches are more frequently used in embedded systems. However, the implementation of fault tolerance techniques at the operating system level may have side effects such as the impact on real-time behavior of the embedded operating system or resource restriction. Therefore, many constraints (especially form performance point of view) should be considered in selecting a recovery technique.

An operating system may crash during several error conditions including: software corruption, hardware malfunction, memory access violation, and executing illegal instructions. Most operating systems immediately stop their operations as soon as they encounter crucial errors in their hardware or software. Kernel panic in UNIX systems is a good example for such behaviors in operating systems.

Among the vulnerable parts of operating systems, *extensions* (which become widespread in commodity operating systems such as Linux) play an important role in the reliability of operating systems. Extensions are optional components which are presented in the kernel address space namely device drivers, network protocols, and file systems. Kernel may include different extensions, and failure in each extension may propagate to the other ones; hence, the dependability of kernel extension is highly important. Extensions cover up to 70% of the operating system source codes, and their error rate is calculated as 3x to 7x more than other source codes in operating systems [1].

Considering the above discussion, the goal of Fault tolerance techniques which are presented in this study is to recover from the transient errors which take place inside the embedded operating system extensions. The common characteristic of these methods is that they do not impose any modification on the base architecture of operating systems. Investigated recovery techniques are *micro rebooting*, *recovery block*, *N-Version Programming (NVP)*, *micro extension*, and *transactional extension*.

The contribution of this study is the evaluation of performance characteristics of well-known recovery methods on the same platform and operating system. The experimental results in this study will provide significant vision for the embedded system designer in using these recovery techniques. For each technique, the recovery time, the CPU utilization, the response time, and the performance penalty are compared with other techniques as well as the baseline operating system.

In this study, from the software point of view, *Embedded Linux* is selected as a target embedded operating system. From the hardware aspect, the modified operating system is executed on an *ARM Cortex A9* CPU, which emulated by *QEMU* [2]. It is noteworthy that the investigated techniques are generic and not architecture specific; thus the results can be regenerated by any other configuration.

To investigate the characteristics of each technique, the totals of 5000 software errors are ignited. The simulation results reveal that the recovery time overhead varies between 0.17% and 0.67%, and the performance overhead varies between 5.81% and 218.65% depending on the techniques.

The remainder of this paper is organized as follows: Section II describes error signaling and the component

 ISBN: 978-1-61208-378-0

isolation support for error confinement. Section III provides technical overview of the investigated techniques. The experimental setup is presented in Section IV and the results are presented in Section V. Finally, Section VI concludes the paper.

II. Error Detection, Containment and Signaling

In this section, error detection, component isolation, and error signaling support for error confinement will be described.

A. Error Detection

An error can be detected by different mechanisms in an embedded operating system. Virtual memory protection, processor exceptions, code checksums, and watchdog timers are some of the well-well known detection methods. To improve the reliability of an embedded operating system, besides the error detection methods, error containment methods should be considered as well. Employing error containment methods leads to the isolation of the erroneous part(s) of an embedded system from the other parts. After detection and containment of an error, as a final step, recovery technique can be applied on the affected component if the error is limited to its inside. The techniques which are under evaluation in this study are placed in the final step.

B. Error Containment

In the following paragraphs, the various isolation mechanisms are discussed in detail.

1) Isolating extensions by code:

A worthy project to isolate component in Linux is Nook [3] [4]. The Nooks isolation mechanisms avoid errors that occur in the extensions in order to affect the kernel. Each kernel extension in Nooks runs in "*light weight kernel protection domain*", which is considered for each kernel extension. Isolation mechanism can provide two main features for a system. The first one is to protect the domain from any manipulation. The other feature is "inter-domain" control transfer.

2) Isolating extensions by virtual machines

"*Virtual Machine*" is another method which isolates the extensions from the rest of a system [5]. In this method, when an extension is called, the unmodified version of that extension is run on its original operating system by a virtual machine. This mechanism allows wide reuse of exiting extensions, without considering the operating system. By running each extension in a virtual machine environment, this method isolates faults produced by faulty extensions. In addition, [6] and [7] utilized virtualization to confine extension in its virtual machine.

3) Isolating extensions by moving them to User-Space

The method introduced in [8] proposes to run extensions as unprivileged user mode. The results of this study reveal that extensions can be isolated without considerable performance degradation.

Besides the above methods, the Micro-Driver introduced new architecture which maintains critical time consuming codes in the Linux kernel and moves the reminder of the extension code to user-mode process [9]. Furthermore, in [10], user-level driver is implemented for Windows NT.

4) Compiler-level extension isolation

The Open Kernel Environment (OKE) project supports fully optimized code to be loaded in the kernel [11]. The OKE enables the restriction modification on the code executing in the Linux kernel. The Decaf Driver is another approach to develop drivers by modern languages such as Java [12]. In Decaf Driver, Linux extensions are converted to Java language and then executed in user mode.

5) Isolating extensions by changing architecture design

In a number of operating systems, a microkernel is implemented instead of using a monolithic kernel. Microkernel only provides simple kernel services. Other operating system functionality is transferred to the user space and does not execute at the privileged level. These architectures intrinsically increase the reliability of the system since each module can be individually controlled. MINIX3 [13], Mach 3.0 [14], Choices [15] and L4 [16] are some of the operating systems that benefit from microkernel architecture.

C. Error Signaling

Exception handling is usually employed to signal errors in user code. In the Linux kernel, the use of exception handling has been explored in [17]. Hence, system designers can write exception handlers to manage errors such as null pointers and invalid op-codes execution in the operating system. This allows designers to develop a flexible and robust technique to handle errors. Generic handlers only print out an error message and stop the operation of the system; however, local exception handlers generate a desirable response and try to recover from failures.

III. Recovery Techniques

The techniques which are introduced in this study can be implemented simply through software approaches on the operating system components. All these techniques are dealing with transient failures. In the following subsection, the architecture of these techniques and how they are implemented in our evaluation will be explored.

A. Micro Rebooting

Considering the terminology of micro rebooting, re-execution of the specific part(s) of an application (not the whole application) is called micro-rebooting. As expected, this technique uses time redundancy to recover errors. Micro rebooting can be applied in both application programs and/or operating system. For the first time, micro-rebooting was employed to recover faulty application components in [18]. The evaluation presented in [18] shows that employing micro-reboot increases the availability by reducing recovery time. This technique also can be used at operating system to recover faulty components [4]. In [15], it is shown that performing micro rebooting on faulty extensions is a simple and effective technique to enhance dependability of operating system.

In our implementation, micro-rebooting mechanism has two parts. The goal of first part is to bring the system and its extensions back into clean state. In this part, we insure that

 ISBN: 978-1-61208-378-0

resources are not taken after they released. In the second part, recovery mechanism runs user-mode recovery agent, which can set recovery policies for extensions before reloading them. Those policies can be written by users in the configuration files. The main task of recovery is to unload extension and load it again. When an error is detected by the detection mechanism, it signals to recovery agent and it runs recovery routine. After reloading the faulty extension by the agent, it signals the application to send its request again.

B. Transactional extension

Transactional extension is another approach to recover systems by using time redundancy. In database expressions, a transaction is a set of operations, which donate a unit of consistency and recovery. Features provided by transactions are isolation, failure atomicity and recoverability [19]. Transactional extension performs transactional operations with four features "ACID". These features are atomicity, consistency, isolation and durability [20]. Durability can frequently be ignored to simplify implementation.

In MARS project, a transactional model was exploited to define the activities of a real-time system [21]. The VINO kernel also used this technique to protect the kernel against misbehaved kernel extensions [22]. In addition, transactional component and micro rebooting are both used in the Choices. The Quicksilver distributed system is another project which exploits transactions [20].

In our transactional extension, before performing any operation, the state of the extension has to be saved. If an error occurs during the transactional operation, the state can be rolled back and the transaction will be aborted. Subsequently, the operation is re-executed. Applying transactional model on extensions leads to performance and space overheads. The need to save extension states before the operation commitment causes space overhead. Moreover, the time overhead is due to perform extra operations. Therefore, the overhead of this technique depends on the granularity of the operations.

There is a main difference between micro rebooting and transactional extension. The micro rebooting reloads extension and re-initializes its internal state. However, the transactional component only rolls back current transaction. Each of these techniques can be used depending on the extension. In general, if an extension has a large volume of data and many internal states, it is more efficient to use transactional techniques since occurring an error may harm the amount of data in micro rebooting; moreover the recovery process impose noticeable overhead. If data loss is not highly important, micro rebooting is a suitable candidate for recovery technique in terms of reducing overhead.

C. Recovery Block

The main recovery block structure is diverse software fault tolerance technique, which is categorized as dynamic techniques. The hardware fault tolerant technique related to the recovery block is stand-by sparing. This technique uses backward strategy to achieve fault tolerance.

In general, recovery block consists of two variants and one acceptance test. The first variant is called primary alternate or primary try block. Another variant called secondary alternates. These blocks are located in the series. In addition, real-time implementation of recovery block includes a software watchdog timer.

In the implementation of recovery block in this paper, the following extension and procedure are implemented:

- Primary extension, Secondary extension (it is equal to the primary), Manager Procedure, Save procedure, Restore procedure, Acceptance test procedure, and Send result procedure.

Our acceptance test is implemented as an application-dependent error detection mechanism such as reasonable check. The acceptance test is unique for two extensions and it includes no fault tolerance approach as it should be simple and quick. Watchdog timer procedure is used to detect irregular behavior such as infinite loop. The manager takes request from application program and saves state and request. Then, it sends request to primary extension. Simultaneously, it also starts a timer. If the response is not returned from primary extension, manager waits until timer trigs an exception. If the deadline is missed, manager unloads the primary extension, restores the state and issue a request. If the deadline is not missed, the manager sends request to acceptance test. According to the result of acceptance test, the manager decides to send the result to application or sends it to secondary extension (in the case of receiving error signals from acceptance test unit). If none of the extensions give correct response, the manager has to send error code to the application program.

D. N-Version Programming

The NVP is one of the well-known design diverse software fault tolerance techniques. The NVP is a static technique in comparison of recovery block. Since a task is executed by some programs, the result is selected among programs results via a majority vote.

In this paper, the NVP is implemented the same as Three Modular Redundancy (TMR). It means that we use three different versions of an extension. The difference of TMR with NVP is that, TMR cannot cover programming mistakes or bugs, because it uses three copies of a program, which are equal, but it can mask other types of errors as well as NVP. The benefit of NVP (with three version of a program) is that it can transparently recover or mask one error. If an error occurs and the detection mechanism detects it during execution of an extension, it sends a signal to voter, and then the voter omits the result of faulty extension. Like recovery block, NVP has a manager procedure which reloads faulty extension. The manager is responsible to take a request from applications and send the result back to them. Thus the application only interacts with one module.

E. Micro Extension

Micro extension is a combinational technique which includes: micro rebooting, transactional extension and user-level isolation mechanism.

The main goal of micro extension is to reduce kernel extension size, and increase reliability of operating system. This is done by moving some parts of extension to user space. In addition, its objective is to recover faulty extensions with the minimum overhead. To reach the goal, new approach is proposed which recovers only some parts of extension. This

 ISBN: 978-1-61208-378-0

technique neither is as fast as micro rebooting nor as slow as a technique which their whole extensions are fully in user space.

It was shown that 65% of extension operation can be moved to the user space [9]. Moving some parts of extension to user space is a kind of isolation mechanism. Micro extension also saves internal state of extension, before doing user space operations, just similar to transactions. At last, it should be noted that recovery of user level application which performs extension operations is similar to micro rebooting.

The difference between micro extensions recovery mechanism and micro rebooting is that micro rebooting unloads and reloads the whole extension without any restoring information; however, micro extension recovery mechanism only destroys and recreates the application which performs user-level operations on behalf of the extension. It should be noted that the extension is not changed any longer. Furthermore, this recovery is transparent from applications which have sent requests for extension. Additionally, this mechanism can restore internal state of extension (which is saved before invoking the extension operation from application) after application failure.

IV. EXPERIMENTAL SETUP

In this section, the experimental setup used in our implementations is presented.

A. Experimental Testbed

1) Operating system

Today, Linux is one of the most employed operating systems in embedded applications which deliver its service through GPL license. The ability to change the kernel in Linux-based operating systems made it possible for developers to customize the kernel by considering customer's demands. A noticeable portion of the introduced techniques is that they use a feature of Linux kernel called *Loadable Kernel Module (LKM)*. According to the above discussion, Embedded Linux is selected as a target embedded operating system in this study. The source code of Embedded Linux is available at [23].

2) Hardware configuration

QEMU as an open source machine emulator [2] is considered in the evaluation. In this paper, Cortex-A9 CPU (ARMv7) and Vexpress-a9 machine are chosen to emulate a system with 128MB of memory. This configuration provides a virtual ARM environment that runs Embedded Linux.

B. Error Activation

In this study, evaluation platform of recovery techniques requires error activation and detection units in order to signal error to error handler. Afterwards, the handler deploys appropriate technique to recover from the errors.

For the evaluation of recovery techniques, 1000 Software error activating experiments were performed for each of the five techniques. Fault model considered in this study to active errors are pointer dereferences, invalid arguments, and bad parameters which randomly injected in the extension. Table 1, depicts the faults model which were injected in the extension and their detection latency. Moreover, the response of system is reported when there is no fault tolerant technique.

TABLE I. FAULT TYPE

Fault location	Response of System	Detection Latency
Command's parameter of System Call	Error code	40960 clock cycle
Address's pointer of System Call	Error code	46336 clock cycle
Pointer of extension's internal function	Kernel panic	77632 clock cycle
Data structure of extension	Application termination and exception	177280 clock cycle
Computation of extension (to create infinite loop)	Kernel hang	200 ms
The application's data which is under control of extension	No signal	---

C. Test Methodology

The goal of this study is to perform a fair comparison among operating system-based fault tolerance techniques. To achieve this goal, a common workload should be considered for all the five techniques. Hence, an arbitrary extension with full controllability is written to explore the techniques considered in this study. Meanwhile, these five techniques can be applied to the real extensions.

The main task considered for the extension is arithmetical operation on matrixes. Three reasons can be enumerated in order to choose such task for extension. The first reason is that in several device drivers in order to increase computation speed, most computations are performed in the driver which is executed with high privilege and at the highest speed. For example, if a driver needs to calculate a parameter, there is no need to perform it at user level. Therefore, our extension can model these behaviors in a proper manner. The second reason is that the vulnerable part of the drivers is their computational part. The last reason is that blocks of data usually are transferred between an extension and an application as in network drivers. In this case, our extension can exchange large matrix with an application. Sorting algorithm is considered for extension task. Because NVP and recovery block need three and two different versions. The Bubble sort, the Insertion sort, and the Selection sort were selected. The primary task for micro rebooting, micro extension, and transactional extension is bubble sort.

To use this extension, a workload is needed to perform computation on matrix and work with them. Therefore, a data intensive application which can work with the extension and perform many computations on matrixes is constructed. This application is considered as the workload to evaluate the techniques. If the extension and the workload are changed, the comparative results cannot be changed a great deal. Hence, we try to report the comparative results (which expressed with percentage).

For measuring time, two mechanisms are used. "*Jiffies*" is used for measuring execution time, which is provided by Linux operating system. In the experiment, one "jiffies" is

 ISBN: 978-1-61208-378-0

equal to one millisecond. For precise measurement, the ARM cycle counter register (CCNT) which is provided by the processor is considered as well.

MR, TR, RB, NVP and ME are abbreviations which stand for micro rebooting, transactional extension, recovery block, N-version programming and micro extension, respectively. In addition WFT shows the average execution time of the application without considering any fault tolerant technique on the operating system.

In this paper, the performance overhead, the recovery overhead, the response time overhead, and the CPU utilization is reported. These parameters allow conducting deeper comparison from performance point of view.

V. Evaluation Results

In this section, the results of employing recovery techniques on the operating system are explored.

Before applying any fault tolerant techniques, the execution time of the application is measured. Moreover in this situation, the response time of the application request is measured. It should be note that these two values are the baseline values and any other result taken from the modified operating system will be compared with these values. It is evident that the QEMU has an impact on the performance, but it can be connived because its affect on all techniques is equal. Table 2 shows the application's execution time in different scenarios.

TABLE II. Application's Execution Time

		Techniques				
	WFT	***MR***	***TR***	***RB***	***NVP***	***ME***
Execution Time(ms)	7423	7855	8702	7897	23654	20004
Standard Deviation	17.0	21.4	41.9	22.1	18.8	21.9

(a) Average execution time without error activation

		Techniques				
	WFT	***MR***	***TR***	***RB***	***NVP***	***ME***
Execution Time(ms)	7423	7908	8734	7948	23695	20093
Standard Deviation	17.0	62.3	88.1	35.1	13.8	20.6

(b) Average execution time with error activation

As Figure 1 shows, the NVP has the maximum performance overhead, but it is not a bad feature. Since one request has to run on three extensions and after voting, the result is returned. Except the NVP, the result of Micro Extension seems incredible. This amount of overhead is related to operating system changing mode for each operation in one request. In fact, for each operation, the kernel extension runs an application in user space by means of API. This result reveals that it is inefficient to use Micro Extension technique when the extension is very computational. Therefore if the role of extension is changed, this can be expected that the performance overhead of Micro Extension technique will be changed as well. It means that the performance overhead is depended on the amount of extension's computational part in this technique. The minimum performance overhead belongs to Micro Rebooting. Transactional Extension has more overhead in contrast to Micro Rebooting and Recovery Block, because it has to save internal state before each transaction.

The response time overhead chart is similar to performance overhead one. It is clear that performance overhead of a workload which is created of several requests is directly related to response time of each request. Figure 2 shows response time overhead.

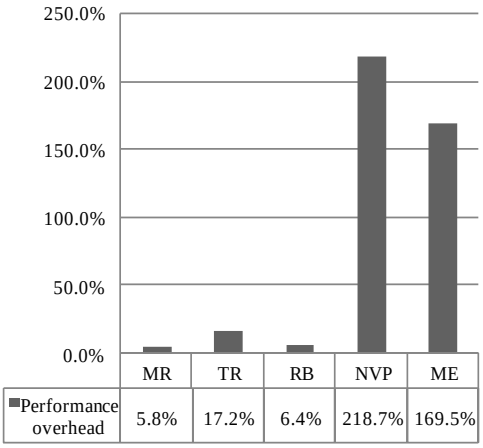

Fig. 1. Performance overhead

Figure 3 illustrates the recovery overhead of each technique. It is expected when a technique forces extra performance overhead, it provides better recovery overhead. The NVP has the best recovery overhead because nothing more is done by the technique when an error occurs. The NVP always masks one error. Since Micro Extension performs an operation in user mode, its recovery overhead is a little more than Transaction Extension. Regarding the CPU utilization, all techniques increase CPU utilization except Micro Extension, which is shown by Figure 4. It also decreases the CPU utilization. It happens because in the Micro Extension, some portion of time is devoted to context switch and transferring data between kernel and user-level part of extension.

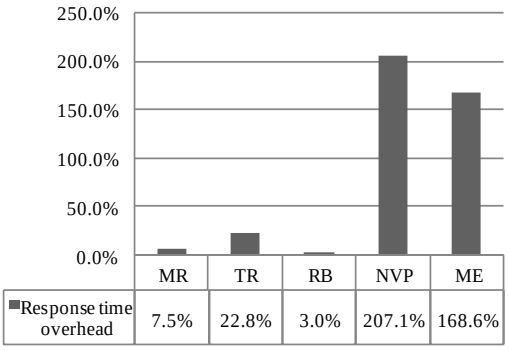

Fig. 2. Response time overhead

 ISBN: 978-1-61208-378-0

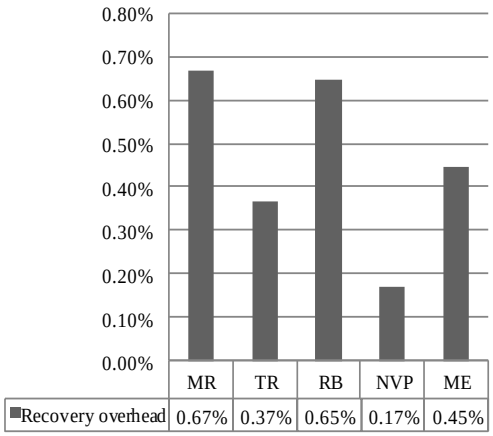

Fig. 3. Recovery overhead

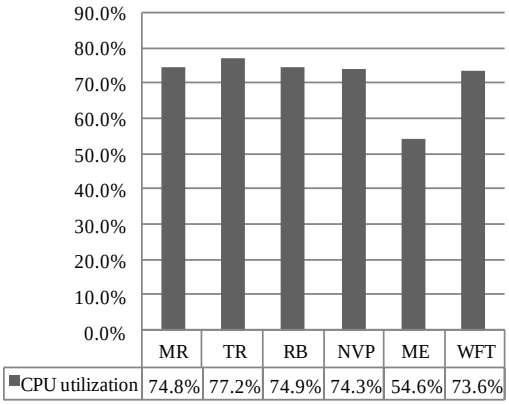

Fig. 4. CPU utilization

At last, the NVP has the best recovery time and the worst performance overhead. Furthermore, the Recovery block has the best response time and the Micro extension has the minimum CPU utilization.

VI. CONCLUSION

An analysis and comparison of five well-known recovery techniques, i.e., micro rebooting, recovery block, N-Version Programming (NVP), micro extension, and transactional extension for an embedded operating system are provided in this paper. These techniques are applied on the operating system extensions without any modification on the architecture of the operating system. This study investigates and compares the characteristic of those techniques on the same platform and operating system from performance point of view. This characteristic leads to an accurate and fair comparison among these methods.

The techniques are implemented on a virtual ARM machine which is emulated by the QEMU under the control of Embedded Linux operating system. The totals of 5000 experiments are made. The experiments results reveal that micro rebooting has the best performance overhead; otherwise, NVP has the worst performance overhead. In addition, the NVP has the best recovery overhead but micro rebooting has the worst one.

The simulation results are as follow: the recovery time overhead varies between 0.17% and 0.67%, and the performance penalty varies between 5.82% and 218.66% depending on the techniques. Additionally, extensions response time, in comparison with the base system, increases between 2.98% and 207.088%. Depending on the techniques, the CPU utilization is confined between 54.63% and 77.24%.

REFERENCES

[1] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, “An empirical study of operating system errors,” Proc. ACM Symp. Operating Systems, vol. 35, Dec. 2001, pp. 73- 88, doi:10.1145/502034.502042.

[2] F. Bellard, “QEMU, a fast and portable dynamic translator,” Proc. USENIX Annual Technical Conference, April 2005, pp. 41-46.

[3] M. Swift, M. Annamalai, B. Bershad, and H. Levy, “Recovering device drivers,” ACM Trans. on Computer Systems, vol. 24, Nov. 2006, pp. 333-360.

[4] M. Swift, B. Bershad, and H. Levy, “Improving the reliability of commodity operating systems,” Proc. ACM Symp. on Operating systems principles, vol. 35, no. 7, Dec. 2003, pp.207-222, doi:10.1145/945445.945466.

[5] J. LeVasseur, V. Uhlig, J. Stoess and S. Gotz, “Unmodified device driver reuse and improved system dependability via virtual machines,” Proc. Symp. on Operating System Design and Implementation, Dec. 2004, pp. 17-30.

[6] L. Tan, et al., “iKernel: Isolating buggy and malicious device drivers using hardware virtualization support,” Proc. IEEE Symp. on Dependable Autonomic and Secure Computing, Sept. 2007, pp. 134-144.

[7] T. Katori, L. Sun, D. K. Nilsson, and T. Nakajima, “Building a self-healing embedded system in a multi-OS environment,” Proc. ACM Symp. on Applied Computing, March 2009, pp. 293–298.

[8] B. Leslie, et al., “User-Level device drivers: achieved performance,” Journal of Computer Science and technology, vol. 20, no. 5, Sept. 2005, pp. 654-664.

[9] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and Somesh Jha, “The design and implementation of microdrivers,” Proc. of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, March 2008, pp. 168-178.

[10] G. C. Hunt, “Creating user-mode device drivers with a proxy,” Proc. of the 1st USENIX Windows NT WS, 1997.

[11] H. Bos and B. Samwel, “Safe kernel programming in the OKE,” Proc. IEEE Conference on Open Architectures and Network Programming, June 2002, pp. 141-152.

[12] M. J. Renzelmann and M. M. Swift, “Decaf: moving device drivers to a modern language,” Proc. USENIX Annual Technical Conference, June 2009.

[13] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, “Construction of a highly dependable operating system,” Proc. IEEE European Dependable Computing Conference, October 2006, pp. 3-12.

[14] F. M. David, and R. H. Campbell, “Building a self-healing operating system”, in Proc. IEEE International Symp. on Dependable, Autonomic and Secure Computing, Sept. 2007, pp. 3-10.

[15] H. H¨artig, M. Hohmuth, J. Liedtke, S. Schonberg, and J. Wolter, “The performance of μ-kernel-based systems,” Proc. ACM Symp. on Operating System Principle October 1997, pp. 66-77.

[16] A. Forin, D. Golub, and B. Bershad, “An I/O system for Mach 3.0,” Proc. USENIX Mach Symp., 1991, pp. 163-176.

 ISBN: 978-1-61208-378-0

[17] H. I. Glyfason, and G. Hjalmtysson, “Exceptional kernel: using C++ Exceptions in the Linux Kernel,” October 2004. Available at: http://netlab.ru.is/exception/KernelExceptions.pdf

[18] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, “Microreboot – a technique for cheap recovery,” Symp. on Operating Systems Design and Implementation, vol. 4, Dec. 2004, pp. 31-44.

[19] J.N. Gray. “Notes on database operating systems,” In R. Bayer, R.M. Graham, and G. Seegmueller, editors, Operating Systems: An Advanced Course, Springer-Verlag, 1979, pp. 393-481.

[20] F. Schmuck, and J. Wylie, “Experience with transactions in QuickSilver,” Proc. ACM Symp. on Operating Systems Principles, October 1991, pp. 239-253.

[21] A. Damm, J. Reisinger, W. Schnakel, and H. Kopetz, “The real-time operating system of Mars,” Operating System Rev. July 1989, pp 141-157.

[22] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith, “Dealing with disaster: surviving misbehaved kernel extensions,” Proc. USENIX Symp. on Operating Systems Design and Implementation, October 1996, pp. 213-227.

[23] Embedded Linux source code, Available at: https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.9.tar.xz

 ISBN: 978-1-61208-378-0