Experiences from Verifying a Partitioning Kernel Using Fault Injection

Raul Barbosa, Johan Karlsson

Experiences from Verifying a Partitioning Kernel Using Fault Injection

Raul Barbosa and Johan Karlsson
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
{raul.barbosa, johan}@chalmers.se

Abstract

This paper describes the usage of fault injection for testing a version of the μC/OS-II kernel which we extended with robust partitioning mechanisms. The implemented mechanisms were tested using a new fault injection plug-in for the GOOFI tool, which aims to provide robustness testing for partitioned systems. We describe the kernel extension and the experiments, with the goal of fault removal, that explore the capabilities of the new plug-in for testing the partitioning mechanisms. The experiments exposed two vulnerabilities in the extension, showing the importance and potential benefits of using fault injection for the assessment of partitioned systems.

Keywords: *fault injection, partitioning kernel, fault removal, embedded systems.*

1 Introduction

Embedded systems have traditionally been implemented by dedicating a computer node to each software component or function. This architecture, which is often referred to as *federated*, has the advantage of providing clear fault containment boundaries in a design. Each software component executes independently on its own processor and resource sharing is reduced to message passing through a communication infrastructure. The need for fault tolerance is satisfied with the introduction of redundant computer systems as well as redundant communication channels. This approach makes it simple to contain hardware and software faults in the computer where they originate.

The main drawback of federated architectures is that they lead to a proliferation of hardware as the number of functions grows. The trend to increase the number of subsystems (designed to add new and enhance existing features) demands a large number of micro-controllers – one per major function. The consequence of such designs is the reliability and cost problems faced by manufacturers of embedded systems. The use of many independent computer subsystems increases the cost of acquisition, space and maintenance, as well as the power consumption. Moreover, a larger number of hardware units leads to a higher fault rate that may reduce the system's reliability.

To address these problems, there is currently a trend to integrate different functions and software components into a common hardware platform with few but powerful processing elements. Such *integrated* architectures have a great potential to reduce cost and improve reliability, since they require fewer hardware components than federated architectures. Furthermore, integrated architectures favor the integration of Commercial Off-The-Shelf (COTS) software in order to reduce development and maintenance costs.

However, to achieve these improvements, it is necessary to equip the system with robust partitioning mechanisms. Such mechanisms prevent faults in the design of one function from disrupting the operation of other coexisting functions [14]. Robust partitioning mechanisms should therefore ensure fault containment within nodes. These mechanisms must prevent processes from writing into each other's memory space – spatial partitioning – as well as ensuring that there is no interference in the time domain – temporal partitioning –, which encompasses both task scheduling and concurrency control.

This paper discusses ideas on the design of fault-tolerant operating systems for embedded applications. The purpose of the operating system is to create a partitioned environment which can be shared by multiple real-time tasks, possibly with distinct levels of criticality and uneven reliability. The principal objective is to facilitate composability within computer nodes, by preventing undesired interactions among software components that share hardware resources.

We describe an extension to the μC/OS-II real-time kernel, named SECERN – meaning *to separate* components from each other. This extension is intended for experimentally assessing techniques for building robust operating sys-

tems. Reusing an existing code base, instead of creating a new solution, has the advantage of making the results more general and focusing the development effort on fault tolerance mechanisms. However, the trade-off is that many design decisions are inherited and may require adaptation to circumstances differing from the original purpose, thereby requiring some verification effort.

We conducted series of preliminary tests of the implemented mechanisms using fault injection. A new fault injection plug-in, aiming to provide robustness testing for partitioned systems, was developed for the GOOFI tool [2, 16]. The plug-in targets the Freescale MPC5554 microprocessor, which is the central element of the experimental platform supported by the present version of SECERN. The set of experiments described in this paper explore the capabilities of the MPC5554 plug-in for testing the robustness of SECERN.

The experiments are conducted according to a methodology of focused fault injection, whose main objective is fault removal, *i.e.*, diagnosis and correction of design faults. It consists of setting up finely controlled experiments in accordance with the system properties that are to be verified. This methodology was applied for verifying the partitioning mechanisms, which should be able to isolate faulty applications to guarantee the correct operation of fault-free partitions.

2 SECERN: An Extension to μC/OS-II

The trend to integrate multiple functions into a single hardware platform has created the need for building strong fault containment around software components. Initiatives such as the standard interface for avionics applications [1] and the AUTOSAR project [6] aim at defining the software infrastructures and, particularly, the operating systems that support this level of fault containment. Since those initiatives target safety-critical systems, a fundamental concern is to ensure that resource sharing can be accomplished in a safe and reliable manner.

We have implemented an experimental prototype of SECERN by extending the μC/OS-II real-time kernel [11]. The source code of the kernel is well documented and freely available for academic purposes, making it a suitable choice for our implementation. The base version of μC/OS-II that we used lacks support for isolating applications from one another and from the operating system, which makes it appropriate for experimentally assessing the SECERN concept.

The extended version of the kernel runs on a computer board featuring a Freescale MPC5554 microprocessor [5], based on the PowerPC architecture. The processor core includes an Memory Management Unit (MMU) which provides, among other services, memory protection. The hardware-specific layer of μC/OS-II was implemented by creating a board support package containing low-level code and macros. The kernel was then extended according to the design principles that are described next.

2.1 Design Principles of SECERN

One of the key modifications to μC/OS-II is the distinction between processes and threads, where each process owns a private address space that groups together one or more execution threads. Each process acts as a container which is usually called a *partition* in Integrated Modular Avionics (IMA) terminology. The architecture of SECERN is depicted in Figure 1.

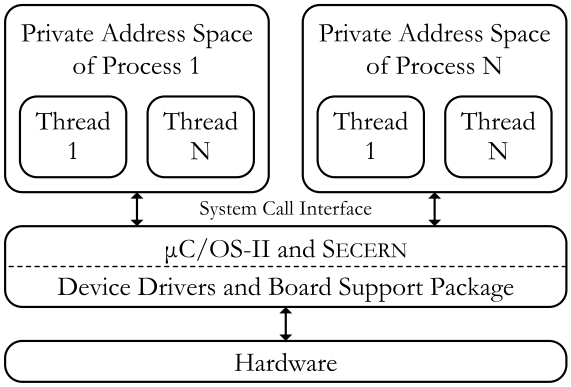

Figure 1. μC/OS-II extended with SECERN.

The private address space of each process is protected by the memory management hardware, which lies between the processor core and main memory. Instructions always generate virtual addresses that are translated by the MMU to physical addresses before a memory operation is performed. During this process, the MMU checks that the application process which is executing has the appropriate access rights – read, write or execute permission for user- and kernel-mode instructions. This feature is used to enforce the appropriate access permissions on all memory pages. For simplicity, a direct mapping is set between virtual and physical addresses, *i.e.*, in practice, no use is made of the address translation feature.

Memory protection is a standard feature of desktop and server computers. However, it is seldom used in embedded real-time systems. One reason for this is that microcontrollers are usually not equipped with the necessary hardware, in order to reduce cost and power consumption. Another reason is the variation in execution time imposed by memory protection and address translation, which is usually optimized for performance rather than predictability.

Typical implementations of memory management hardware make use of a Translation Look-aside Buffer (TLB) for improving the performance of address translation and

memory protection. A TLB is a very fast cache which contains a small number of entries; each entry specifies the virtual and physical addresses where a memory page starts, the size of the page and the access rights. This cache reduces the time overhead of the MMU but there is a large penalty for memory accesses which are not matched by any TLB entry. In this case, which is called a TLB-miss, a processor exception is raised to allow the system software to update the TLB. This may become an issue, since interrupts are generally unwanted in real-time systems and make it more difficult to determine the Worst-Case Response Time (WCRT) of applications.

To deal with this problem, the memory protection routines of SECERN are designed to update the TLB during context switches. The approach is to insert in the TLB the pages that belong to a process before running that process, thereby preventing TLB-misses. This, in turn, simplifies the response time analysis for hard real-time tasks. Nevertheless, this method adds an overhead to context switches.

The time needed for a full context switch without updating any TLB entries is slightly below 10 μs (for saving the numerous PowerPC context registers, updating kernel structures and loading the registers of the next task). Considering a typical embedded application, requiring between 4 and 8 pages of memory, context switching would take between 31 and 53 μs. This overhead should be carefully examined when considering performance demands, as it is common for real-time operating systems to switch context in less than 10 μs. Nevertheless, when memory protection is used, this increased time is a trade-off rather than a penalty. Without updating the TLB, a process may cause in the worst case one TLB-miss for each page. This is more expensive than doing the update during context switches and generates execution time jitter.

Introducing memory protection has implications on the design of the system call interface, since it rules out the use of the *branch and link* instruction for calling system services. Instead, service requests are made through the system call interrupt. This process is made transparent to applications by implementing the low-level details in a system library – a common approach in operating system designs.

The system call mechanism is used by applications to request kernel services and to reach device drivers. For this reason, it must be robust in order to prevent application errors from propagating to other parts of the system. This is often a problem, as experimental studies have shown that many operating systems contain vulnerabilities in functions provided by the system call interface [10], *e.g.*, crashing the system when given exceptional input parameters.

Another problem is that the system call mechanism must be able to enforce access policies, in order to control the services that each partition has the right to access. Some authors propose the usage of sandboxing as the means to protect the system call mechanism [12, 13]. This technique consists of interposing the access to system calls with a filter that enforces a given policy. For real-time kernels, this technique must be implemented as efficiently as possible.

We took a simple approach to implementing system call protection. The kernel provides the partition's ID to the system call handler. The caller ID can be checked by the drivers and by any kernel services to enforce an access policy. It is also possible to check the parameters to the system call interface and report an error of the partition that executed the call. This would act as an additional error detection mechanism.

One of the limitations of the current version of SECERN is that it does not introduce mechanisms for temporal partitioning. μC/OS-II has a priority-based preemptive scheduler that executes always the task with the highest priority which is ready to run. This means that a high priority task may prevent lower priority tasks from executing, if it fails to release the Central Processing Unit (CPU) on time. On the other hand, this ensures that the highest priority task is never disturbed by any other task (a limited form of temporal partitioning). In the fault injection experiments described in this paper we focus on the behaviour of the highest priority task.

2.2 Error Detection and Fault Handling

In addition to memory protection and checking the system caller ID, our kernel extension makes use of processor exceptions to detect errors. Moreover, it allows application-specific checks to notify the kernel of errors. Many techniques for creating application-specific checks are available in the literature and the kernel provides the means for such checks to report errors. When any of these error detection mechanisms is triggered, SECERN handles the error in one of two central exception handlers:

- *Unrecoverable condition.* An error is detected which may be caused by a hardware problem or by a fault in the operating system itself. All exceptions which cannot be safely considered to be caused by one partition are unrecoverable. An example is an invalid memory access attempt by the kernel. The currently implemented version enters an infinite loop in case of an unrecoverable condition. It would be possible, for instance, to restart the kernel, check the consistency of the hardware and restart all tasks.

- *Recoverable condition.* The detected error is confined to a single process (*i.e.*, partition) and it is possible to delete that process and continue executing. Examples of such errors are invalid memory accesses or invalid instructions executed by a partition. In this case, SECERN deletes all threads belonging to the process

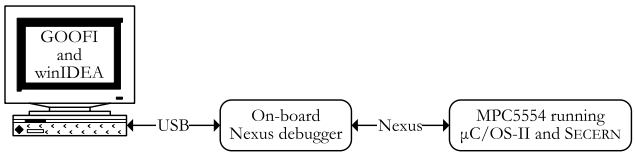

Figure 2. Evaluation platform for μC/OS-II and SECERN.

and resumes execution. In hard real-time systems it is fundamental to ensure that fault handling activities are processed in a timely manner (in this case one would have to determine the time needed to delete the faulty process).

3 Robustness Testing for Partitioned Systems

We have extended the GOOFI tool [2, 16] with support for injecting faults into the Freescale MPC5554 microprocessor. The new fault injection plug-in is based on an existing plug-in which provides support for the MPC565 processor. The experimental setup consists of a desktop computer, with GOOFI and the winIDEA development environment, controlling an MPC5554 development board [8]. The development board includes an on-board Nexus debugger. Figure 2 depicts the experimental platform.

The MPC5554 fault injection plug-in is capable of automatically injecting bit-flips into processor registers and memory locations. It allows the user to define a range of code addresses where the execution can be stopped for injecting a fault. In each experiment the tool selects one random code address to set a breakpoint for fault injection. The tool then chooses a resource randomly (register or memory location) and one of its bits to inject the bit-flip once the breakpoint is reached.

This plug-in, unlike its predecessors, does not collect the sequence of instructions executed in the fault-free experiment (*i.e.*, it does not create a program trace). Doing so is a time-consuming procedure, since the processor needs to be stepped in order to determine the sequence of values of the program counter. Due to the large number of instructions executed by the kernel, the application processes, the idle task and other system tasks, the stepping process for the reference experiment would take too much time.

To deal with this, the tool allows fault injection experiments to be made without a program trace. This is achieved by choosing a random address from the entire range of user-defined addresses. Since that address might not be reached once the program executes, there are a number of experiments in which a fault is never injected and the outcome is exactly the same as that of a fault-free experiment. Such experiments are simply discarded during analysis and classification.

To provide fault injection for a partitioned environment, the tool is capable of monitoring the execution of the operating system and collect the output of multiple tasks. The user can define the output address of multiple workloads, so that the results produced by tasks can be collected and classified. Moreover, the tool can set breakpoints for monitoring the activation of the two central exception handlers described in the previous section, in order to monitor the operating system. The activation of breakpoints and the output data are saved to a database for analysis.

Regarding the workloads used as operating system tasks, we use cyclic programs that execute some computations on input data and delay themselves until the next iteration. Figure 3 shows the typical structure of the main routine of a workload thread. The output breakpoint can be set to the address before the call to `OSTimeDlyUntil()`.

```
void thread(void *pdata)
{
  INT32U next_time, period = 20;

  next_time = OSTimeGet();

  while(TRUE)
  {
    getInput();
    computeOutput();
    OSTimeDlyUntil(next_time += period);
  }
}
```

Figure 3. Main routine of a workload thread.

4 Focused Fault Injection

One may conduct fault injection experiments with the purpose of fault forecasting or fault removal. Fault forecasting experiments aim to estimate diverse measures of dependability and to gain a better understanding of how a system (or one particular component) behaves in the presence of real faults. Such experiments are useful for comparing alternative components with regards to their dependability, for identifying a system's dependability bottlenecks, for characterizing a system's dependability, etc.

The goal of fault removal experiments is to identify flaws in the design or implementation of a component or a system, so that they can be corrected. To achieve this, one places the focus of experimentation on exercising specific parts of the system with suitable types of faults (which the system is required to tolerate). This form of fault injection is suitable for testing fault tolerance mechanisms and is therefore helpful for the verification of computer systems.

Fault forecasting is a very frequent objective of fault injection practitioners. Researchers often adopt this method for experimentally validating new techniques, *e.g.*, by determining the coverage provided by an error-detecting mechanism or the effectiveness of a recovery strategy. Taking a broader perspective, there have been research efforts to promote the use of measurement theory for estimating dependability [3] and to define methods for benchmarking the dependability of computer systems [9]. Dependability benchmarks aim, among other things, to guide the development effort (*e.g.*, by finding weaknesses in an architecture) and to assist buyers in deciding among competing off-the-shelf components.

Nevertheless, fault removal is also vital for many, if not most, buyers of COTS software. Consider an example where a system integrator intends to use an off-the-shelf operating system for building a given application. The selection process is influenced by numerous factors, including technical findings – such as results of dependability benchmarks – and management decisions – based on each vendor's credentials, guarantees in terms of long term support, cost issues, etc. We can identify two risks here. First, the selected operating system might not be the most dependable among the available choices. Second, regardless of the choice, it may require adaptation to a specific hardware platform and it could contain design or implementation defects. Consequently, system integrators would be interested in coming back to suppliers with problematic test cases that require attention.

In this paper we adopt fault injection as the means to find such test cases. We are interested in finding and removing vulnerabilities in SECERN – particularly those related to partitioning. To this end, we begin by describing a methodology for fault removal in partitioned systems and then present the results of fault injection experiments targeting our experimental platform.

4.1 Methodology

A fault injection experiment with the objective of fault removal has two principal outcomes: either the system fails to cope with the fault that is injected (*e.g.*, the operating system crashes) or the service provided by the system is classified as correct. This classification requires sufficient data to be collected during the experiments, so that we can determine whether or not the system fails to handle any faults. If so, those faults can be regarded as counterexamples, *i.e.*, scenarios where one or more system properties are violated.

Naturally, the faultload must be representative of faults that the system is required to tolerate. On the one hand we wish to test systems extensively, in order to identify as many existing defects as possible. On the other hand all counterexamples should be meaningful, *i.e.*, they should only locate actual defects rather than calling our attention to situations which the system is not supposed to handle. To achieve this, we adopt a methodology of focusing fault injection experiments in accordance with the system properties that are to be verified.

The concept of focused fault injection has been used in the past for testing distributed systems [15]. We take a conceptually similar approach targeting the verification of node-layer fault tolerance mechanisms. Our goal is to verify that SECERN prevents application errors from propagating to the operating system and to other applications. We are therefore searching for vulnerabilities in the software related to partitioning mechanisms, *e.g.*, the low-level code that controls the hardware. Nevertheless, one should not exclude the possibility of finding hardware design faults such as those reported by Intel [7], affecting the MMU of recent microprocessors. The fault injection experiments were designed by taking the following steps:

- *Configure the workloads in a relevant manner.* We configured the system to execute two processes, each one with a single thread. The two threads executed, in an infinite loop, a data processing routine and released the CPU until the next iteration. The tasks executed with sufficient frequency to force context switches among them at intermediate points of the execution (of the low priority thread).

- *Inject faults that mimic application errors.* The tool injected bit-flips in the context registers (*i.e.*, processor registers that are saved during context switches) of the lowest priority task. Bit-flips are not representative of software faults but, nevertheless, they are representative of faults that the system must handle. The tool was configured to inject faults during the execution of any instruction of the low priority thread.

- *Collect sufficient data to classify experiments.* During each experiment we collected the output of both tasks and monitored the activation of the two central exception handlers described earlier (to infer whether the operating system had crashed).

- *Classify the outcome of the experiments.* We analyzed the data resulting from the experiments in order to check if partitioning had been violated. First, the output of the high priority task was compared to that of a fault-free reference experiment. Any difference in the results indicates a partitioning violation. Second, the activation of the unrecoverable exception handler indicates that the operating system had crashed. Third, experiments where the execution ended at a different instruction address than the expected one are caused by an undetected system crash.

- *Examine experiments that expose counterexamples*. Faults that cause the operating system to crash, the high priority task to produce wrong/missing output or the high priority task to be deleted are classified as partitioning violations. For these experiments one must examine the fault which was injected (the instruction where the bit-flip was injected and the resource affected), since it exemplifies a situation which is not properly handled. Essentially, the question is to understand what led a fault injected in the low priority thread to affect other parts of the system.
- *If necessary, instrument the code and document test cases*. We can manually instrument the code of the workloads to mimic as closely as possible any fault that exposes a counterexample. This optional step can be useful, for example, when a system integrator finds problematic test cases using fault injection; the system integrator would prefer to send test cases consisting of instrumented programs to the supplier of the operating system, rather than sending the fault injection tool and the fault definitions. Moreover, in our case this serves as a way of validating the fault injection tool.

4.2 Results

We present the results of a campaign consisting of 284 fault injection experiments. Each of the two threads executed a workload consisting of a wavelet transform, which takes an array of input data an produces an output array containing the result of the transform.

In our setup it takes 1min 12s to run a reference experiment for collecting the results of a fault-free execution. Each fault injection experiment takes, in average, 1min 25s. Since we do not collect the program trace (*i.e.*, the sequence of instructions executed during the reference experiment), we must set the fault injection breakpoint without being certain that it will be reached.

Table 1 shows that the fault injection breakpoint was reached, in this set of experiments, in 67 occasions. In the remaining 217 experiments the fault injection breakpoint was not reached and this means that no fault was injected.

We analyzed the 67 experiments where a bit-flip was actually injected to determine whether it was correctly handled. As explained earlier, the classification process takes into account the activation of the centralized exception handlers (recoverable and unrecoverable) and the output of the tasks to determine whether or not the fault was handled. In this case we consider only the output of the high priority task, since we are injecting faults in the low priority task. Table 2 shows the classification of the fault injection experiments.

As we can see in Table 2, the operating system crashed once and the high priority task produced wrong results in three occasions. One of the wrong outputs occurred in the same experiment where the operating system crashed (which made it impossible for the task to continue executing). Thus, we found three experiments where the system failed to handle a fault in the context of the low priority task. One fault led the entire operating system to a crash and two faults caused the high priority task to produce incorrect results. These faults were carefully examined since they exposed flaws in the system.

4.2.1 The Context Switch Flaw

The fault that led the operating system to a crash was injected into processor register R1, which is the stack pointer. At a certain point of the execution of the low priority task, a bit-flip changed the stack pointer from $40007F08_{16}$ to $44007F08_{16}$. In practice, this meant that R1 no longer pointed to the top of the low priority thread's stack and now pointed to an unused memory address.

We used the debugging environment to manually inject a similar fault and observe the sequence of events that then took place. Rather than using the stack pointer, the low priority task was executing a part of the main loop when a context switch occurred. At this point, the μC/OS-II kernel started to save the context of the task to the top of its stack – the approach that it is designed to take. The problem was that the stack pointer no longer pointed to the correct address. Thus, the kernel attempted to write the context of the task to address $44007F08_{16}$. This memory area was unused and therefore not listed in the TLB, thus causing a TLB-miss. In our design, a TLB-miss caused by kernel code is an unrecoverable condition.

The code of the low priority task was manually instrumented to execute correctly for two seconds, corrupt the stack pointer and enter an infinite loop (to wait for a context switch). Figure 4 shows the instrumented code.

This fault showed that our extension to μC/OS-II failed to provide perfect partitioning due to an inherited design decision. Since μC/OS-II saves the context of tasks on the top of their own stack, it is possible for a task to corrupt the stack pointer and cause the kernel to write onto an erroneous memory location.

There are numerous possible solutions to remove this partitioning defect. We chose to add a stack pointer check during context switches. The task control block of all tasks (a kernel structure which stores important task information) contains the location and size of each task's stack. We added a check to verify, before saving the context, that R1 points to a memory location in the task's stack and that there is enough space to write all context registers. After modifying the context switching code we executed the test case in Figure 4 and verified that the fault had been removed.

No. of Experiments	Breakpoint Reached	Breakpoint Not Reached
284	67 (23.6%)	217 (76.4%)

Table 1. Activation of the fault injection breakpoint.

Experiments	Operating System		High Priority Task		
	Operational	Crashed	Correct Output	Wrong Output	Deleted
67	66	1	64	3 (2+1)	0

Table 2. Outcome of the fault injection experiments.

```
void thread(void *pdata)
{
  INT32U next_time, period = 20;

  next_time = OSTimeGet();

  while(TRUE)
  {
    // two seconds after startup
    if(next_time > 200)
    {
      // set stack pointer to 0x44007F08
      __asm__ (" lis  %R1, 17408      ");
      __asm__ (" addi %R1, %R1, 32520 ");
      while(TRUE){ }
    }

    getInput();
    computeOutput();
    OSTimeDlyUntil(next_time += period);
  }
}
```

Figure 4. Manual instrumentation of the low priority thread to corrupt the stack pointer and wait for a context switch.

4.2.2 The Configuration Error

The two experiments that caused the high priority task to produce wrong results injected a fault into registers R6 and R29. These faults were injected at a point of the execution where these registers were being used to calculate memory addresses for write operations. The instructions that executed subsequently attempted to write into a page which was shared by the two tasks, containing code and data belonging to a shared floating point library.

The issue here was that there were several pages erroneously configured with write permission for all tasks. The initialization sequence inserts into the TLB the pages that are listed permanently (kernel and shared libraries). An inspection of this sequence revealed that the pages were configured with full permissions for all tasks, even though they should be only readable and executable. In this case, a test case would be as simple as instrumenting the code of the low priority thread to write into those addresses. This configuration error was solved by giving only read and execute permissions on the library pages to all tasks.

4.3 Discussion and Limitations

These experiments demonstrate the potential of fault injection as a fault removal technique for partitioned systems. Despite the small number of faults injected and examined in this paper, it was possible to find and correct two implementation flaws. However, we would have to conduct many more experiments to exhaustively test the mechanisms included in the extended real-time kernel.

Moreover, a limitation of these experiments is that we only observed the output of the high priority task in the value domain, *i.e.*, the time when task produced its output was not monitored. Thus, temporal partitioning was only examined indirectly, by monitoring whether or not the high priority task produced correct results at some point in time.

We have tested the robustness of the implementation in the presence of bit-flips in the context of one process. Even though this is a type of fault that the system must tolerate, bit-flips in CPU registers and main memory are mostly representative of transient hardware faults. An exhaustive test of the kernel extension should take into account software faults. These can be injected using software fault emulation operators [4].

5 Conclusion

This paper presented SECERN – an approach to provide partitioning and fault tolerance for real-time kernels. SECERN includes several mechanisms to confine errors to the applications where they originate. These mechanisms are necessary for creating a partitioned environment which can

be shared by multiple real-time tasks, possibly with distinct criticality.

The partitioning mechanisms were implemented as an extension to the μC/OS-II real-time kernel. The current version of the extension uses memory protection, processor exceptions, system call protection and application-specific checks to detect errors. These techniques were implemented while taking into account that they must respect the requirements of real-time tasks, *i.e.*, they must introduce low overhead and if possible no execution time jitter. The extended kernel was developed for the Freescale MPC5554 microcontroller.

A new fault injection plug-in was developed for the GOOFI tool, targeting the MPC5554 microprocessor and aiming to provide robustness testing for partitioned systems. We conducted a series of fault injection experiments using the tool for testing the kernel extension. These experiments were conducted according to a methodology of focused fault injection, with the goal of diagnosing and removing design faults.

The experiments exposed two vulnerabilities in the kernel extension: one related to configuration management, where some memory pages were marked as writable for all processes while they should only be readable and executable; and one related to an inherited design decision regarding context switches which is unsuitable for partitioned systems. Even though the tests are not exhaustive, they show the importance and potential benefits of using fault injection for the assessment of partitioned systems.

Acknowledgements

The authors wish to acknowledge Jorge Alçada, Ding Xie and Chi Zhang for their contribution to the development of the kernel extension and the fault injection environment described in this paper. This work was partially supported by the Swedish National Aviation Research Programme (NFFP) project number S4207, and by the Saab Endowed Professorship.

References

[1] Aeronautical Radio, Inc. ARINC specification 653-1: Avionics application software standard interface, Oct. 2003.

[2] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. GOOFI: Generic object-oriented fault injection tool. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN 2001)*, pages 83–88, July 2001.

[3] A. Bondavalli, A. Ceccarelli, L. Falai, and M. Vadursi. Foundations of measurement theory applied to the evaluation of dependability attributes. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2007)*, pages 522–533, June 2007.

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

[5] Freescale Semiconductor, Inc. *MPC5553/MPC5554 Microcontroller Reference Manual (Rev 4.0)*, Apr. 2007.

[6] H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J.-L. Maté, K. Nishikawa, and T. Scharnhorst. AUTomotive Open System ARchitecture - an industry-wide initiative to manage the complexity of emerging automotive E/E architectures. In *Proceedings of the 2004 International Congress on Transportation Electronics (Convergence 2004)*, pages 325–332, Oct. 2004.

[7] Intel Corporation. *Intel® Core™2 Extreme Processor X6800 and Intel® Core™2 Duo Desktop Processor E6000 and E4000 Sequence: Specification Update*. Document No. 313279-026, May 2008.

[8] iSYSTEM AG. *EVB-5554 Evaluation and Development Kit for Freescale PowerPC MPC5554 Microcontroller (User's Manual)*, July 2007.

[9] K. Kanoun and L. Spainhower, editors. *Dependability Benchmarking for Computer Systems*. Wiley, 2008.

[10] P. Koopman, K. DeVale, and J. DeVale. *Dependability Benchmarking for Computer Systems*, chapter Interface Robustness Testing: Experience and Lessons Learned from the Ballista Project, pages 201–226. Wiley, 2008.

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

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

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

[14] J. Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical Report NASA/CR-1999-209347, NASA Langley Research Center, June 1999.

[15] S. Tao, P. D. Ezhilchelvan, and S. K. Shrivastava. Focused fault injection testing of software implemented fault tolerance mechanisms of Voltan TMR nodes. *Distributed Systems Engineering*, 2(1):39–49, Mar. 1995.

[16] J. Vinter, J. Aidemark, D. Skarin, R. Barbosa, P. Folkesson, and J. Karlsson. An overview of GOOFI – a generic object-oriented fault injection framework. Technical Report 05-07, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, 2005.