High Availability in Microkernel-Based Environments

Mihai Carabaș, Larisa Grigore, Sofia Neață
Automatic Control and Computers Faculty
University POLITEHNICA of Bucharest
Emails: {mihai.carabas,larisa.grigore,sofia.neata}@cti.pub.ro

***Abstract*—Allowing applications to survive hardware failure is an expensive undertaking, which generally involves reengineering software to include complicated recovery logic as well as deploying special-purpose hardware. The project implements mechanism for failure detection, prediction and system recovery. The system design approach and associated service implementation ensures that a certain level of operational performance will be met. VMXL4 microkernel supports High Availability systems, providing secure domains isolation of systems and performance monitoring features. Based on these features, we created an architecture for monitoring the system at different levels (intrasecure domain and intersecure domain).**

***Index Terms*—High Availability, microkernel, L4 kernel, secure domains, performance monitoring, fault management**

I. Introduction

High Availability refers to a system or component that is continuously operational for a desirably long period of time. The system design approach and associated service implementation ensures that a certain level of operational performance will be met. Availability can be measured relative to *100 percent operational* or *never* failing. Availability is usually specified over a certain period of time week, year, etc. A widely-held but difficult-to-achieve standard of availability for a system or product is known as *five 9s* (99.999 percent) availability.

Adding more components to an overall system design can undermine efforts to achieve high availability. Complex systems inherently have more potential failure points and are more difficult to implement correctly.

A failure occurs when the observed behavior differs from the expected (e.g. a component does not respond within a certain time frame). The paper proposes a solution for implementing mechanisms of failure detection and system recovery.

VMXL4 microkernel supports High Availability systems, providing secure domain isolation of systems and performance monitoring features. Based on the fact that a secure domain failure is not able to influence the execution of the other secure domains running on the same machine, the high availability module handles only the possible failures inside a secure domain. It is devided into two submodules: intrasecure domain high availability module (handling thread failures such as dead threads, segmentation fault, infinite loop, deadlocks, processor overload etc.) and intersecure domain high availability module (handling operating system failures such as kernel ops and kernel panic).

This paper proposes a High Availability interface witch provides the following features:

- Detection - the fault or some failure conditions are determined using performance monitoring microkernel features
- Diagnosis - analyzes de cause of the fault
- Isolation - the rest of the system is protected from the fault
- Recovery - the system is adjusted or re-started so it functions properly; it involves restoring or delaying the thread execution
- Repair - a faulty system component is replaced; it involves thread or secure domain restart.

The isolation, recovery or repair levels depend on the failure's gravity and the source of failure: it may need to consider the faulty thread, the faulty thread's address space or the entire secure domain.

The paper is structured as followes: Section II presents the Related Work in the domain of High Availability, Section III the Environment we have used for testing and developing, Section IV the Architecture of our proposed solution, Section V the Implementation details, Section VI the problems we have encountered and Section VII presents our Future work and the Conclusions of this paper.

II. Related Work

Software high availability (HA) developed over a microkernel based operating system is not a new concept. QNX [1] is a POSIX, real time embedded microkernel OS and offers great fault isolation and dynamic upgradeability not limited to application level but extended to all elements in the system. Because everything outside the micro-kernel exists in memory protected user space, this protection exists not only for applications but also includes file systems, protocol stacks and even device drivers. In this model, faults are limited to the process in which they occur — device drivers, filesystems, stacks and applications are all processes in the micro-kernel model). To supplement the high availability inherent to the QNX architecture, QNX Neutrino [8] offers a high availability framework whitch enables developers to create custom recovery scenarios. A smart watchdog monitors process status via heartbeating and in the event of a process failure, user defined recover scenarios are executed.

Our HA approach considers the posibility of predicting the program faults before they even occure. In this sense failure detection and localization, several researchers have studied various types of program spectra to predict program execution behavior.

Podgurski et al. [11], [10], [2] present a set of techniques for clustering program executions. Bowring et al. [5] introduce a technique based on Markov models to distinguish failed executions from successful executions using branch coverage information. Haran et al. [6] present several methods for classifying execution data as belonging to one of several classes. Brun and Ernst [12] identify dynamically discovered likely program invariants to reason about program execution behavior. Agrawal et al. [4] and Jones use statement coverage information to identify likely causes of failures. Chen et al. [7] keep track of components exercised during executions to pinpoint faulty components. Santelices et al. [9] empirically evaluate the performance of several types of program spectra in locating defects. Burcu et al. [3] use of a novel, hardware performance counters-based program spectrum to identify likely causes of failures.

Our goal is to combine the HA features offered by microkernel operating systems like QNX Neutrino with the posibility of predicting when a software failure will occure based on hardware counters metrics. In this way we can take recovery actions before the failure occures and not after that (like in heartbeating monitoring approach).

III. Environment

The system over which we implemented the high availability library is based on a L4 microkernel from Virtual Metrix (VMXL4). VMXL4 is a general purpose, high performance microkernel which implements the mechanisms for performance management. It provides a minimal layer of hardware abstraction on which various operating system personalities can be built (through paravirtualization). This abstraction includes interfaces for IPC (message passing Inter-process Communication), address space (memory) management, interrupts, virtualized interrupts and managing system performance resources. It isolates each component in the system from effects of programming errors or malicious code contained in other components. Therefore the monitor is isolated and can't be corrupted. In case something happens with the rest of the systems, the monitor can take appropiate actions.

VMXL4 applies the principle of minimality by providing mechanisms as opposed to services. As such, a feature is included in the microkernel only if it is impossible to provide that service outside the microkernel without sacrificing security, or if including the feature in the microkernel it provides significant benefits without increasing the complexity of the microkernel. For example, there are 12 system calls in this microkernel comparing to Linux where are over 300. Therefore it can be guaranteed that the microkernel is bug free. VMXL4 utilizes a capability-based security model, where a capability (also called a key) is a un-forgeable token of authority. Protection domains and security policy are implemented through capabilities. All IPC depends on having the capability to send to a particular destination.

A unique aspect of VMXL4 is the concept of Performance Monitoring. The goal of Performance Monitoring is collect different type of statistics (cycles, branch count, etc) in realtime. This is done using a generic interface for the hardware mechanisms for collecting CPU information. The interface is implemented through the PerfMon syscall. The microkernel uses an abstraction for hardware counters called a PMU, Performance Monitoring Unit. The PMU consists of PMD (Performance Monitoring Data) - hardware data registers, PMC (Performance Monitoring Configuration) - hardware configuration registers, a list of associations between PMDs and PMCs. Therefore, it offers platform independence and allows multiple threads to use the Performance Monitoring facilities. Using PerfMon to collect different counter values, we are able to detect the system state (eg: infinite loop).

The VMXL4 microkernel is written in C++ and the projects in user space are written in C. The build system is based on Scons building suite, written in Python.

The platform we run on the VMXL4 microkernel is based on an ARMv7 CPU architecture. We use a BeagleBoard as our hardware platform. The BeagleBoard is a low-power, low-cost hardware single-board computer which is equiped with Texas Instrument's OMAP3530 system-on-a-chip. OMAP3530 contains an ARM Cortex-A8 processor, with the ARMv7 instruction set. Alternatively, we used an ARM emulator from QEMU on top of which we have run the microkernel succesfully.

IV. Architecture

The VMXL4 microkernel includes a virtualization mechanism that supports running multiple secure domains, as described in the previous chapter. The main advantage of having independent secure domains is that a failure inside a secure domain cannot affect any another secure domain running on the same machine, at the same time. Thus the problem of building a high available system (that is a system able to prevent, detect and repair failures) can be solved by building a high available secure domain. The microkernel must provide mechanisms that maintain the health of the threads inside a secure domain (intrasecure domain high availability module) and the health of the services responsible to ensure associated operating system functionalities (intersecure domains high availability module). Using this concepts, we have designed the high availability infrastructure, described bellow.

Intrasecure domain high availability module is used for monitoring address spaces modifications, invalid accesses and thread execution. Its functionality is provided using a centralized architecture: the central monitoring thread and multiple worker threads (see Figure 1). Each worker thread is responsible with predicting and/or detecting one or more thread failures (dead threads, segmentation fault, infinite loop,

deadlocks, processor overload, etc.). It is also responsible with notifying the central monitoring thread about the (possible) failure. Based on this notifications, central monitoring thread must take a decision and execute the necessary actions in order to recover the faulty thread (restart, restore or delay execution). In case of memory errors or interdependent threads may require to recover the entire address space. Thus, the central monitoring thread acts as a reincarnation server (see Figure 2).

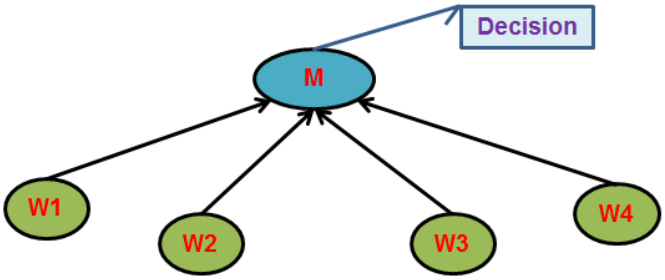

Figure 1. Intrasecure domain workers

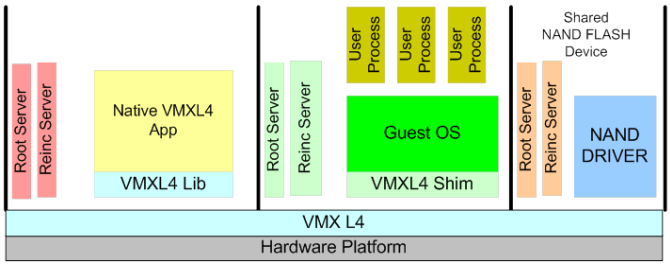

Figure 2. Intrasecure domain high availability module

Intersecure domain high availability module monitors the activity of the secure domains. It must be placed in a separate secure domain (called SecDom0) and it has a similar architecture with intrasecure domain module (see Figure 3). Unlike the faulty threads, recovering a faulty secure domain actually means restarting it. The decision of not restoring a secure domain relies on the high costs of saving its state (the content of the address spaces and threads' state). This repeated operation may result in a system overload that may damage the system's health instead of healing it.

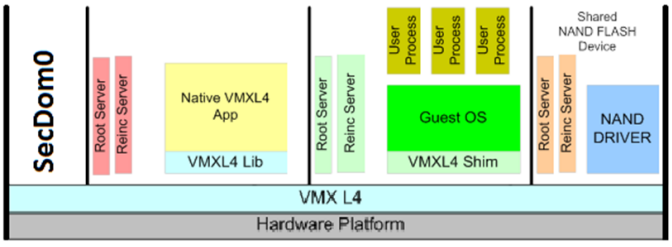

Figure 3. Intersecure domain high availability module (SecDom0)

V. Implementation

As described in the architecture section, we are trying to offer high availability at two levels: intrasecure domain (the monitor runs in the same secure domain as the monitored processes) and intersecure domain (the monitor is in another secure domain).

A. Intrasecure Domain High Availability

1) High Availability Library: We designed and implemented a high Availability module that is available throug a library called High Availability Library that provides an API for thread monitoring. The functions below are available:

```
/* Start HA module. */
int HA_start(void);

/* Stop HA module. */
int HA_stop(void);

/* Checks if HA module is started. */
int HA_is_started(void);

/* Monitor a thread. */
int HA_monitor(L4_Word_t, L4_ThreadId_t,
                          void *, int);

/* Stop monitoring a thread. */
int HA_unmonitor(L4_Word_t);
```

Using the first two functions the HA module can be started and stopped. *HA_monitor* is used by a thread to add another thread or itself to the HA module list of monitored threads. When a thread decides that another one will be monitored it is named *supervisor*. *HA_unmonitor* can be used only by the related supervisor.

2) Monitor thread and communication with worker threads: The monitor thread is responsible with receiving notifications from workers threads and decide what to do accordingly to those messages. Worker threads send a notification when they realize something wrong happened with one of the monitored threads.

The communication between the monitor and the workers is done through a queue where workers put messages serving as producers and monitor has the consumer role. The queue is used with a mutex such as only a thread can access the structure at a certain moment and a condition variable to signal when notifications are available in the queue. A notification is a structure through which a worker announces monitor that one of the monitored threads doesn't work properly or is dead. It contains a label that signals the type of failure (deadlock, infinite loop or the thread isn't alive) and the global identifier of the thread (a thread has two identifiers: a global identifier, unique in the system, and *L4_ThreadId_t* identifier unique in the secure domain).

Monitor thread is running in root space and workers in other space address. Root space is the address space where root task, the thread responsible with resources allocation,

and the pager, the thread handling page faults, are running. In consequence, if something wrong happens with the monitor, the whole secure domain is compromised. We chose another address space for the worker threads in order to protect the root space if something happens with the workers. We chose this design, with multiple worker threads, to allow an easy extension of the architecture and to avoid overloading the root space.

3) Worker threads: In the initial design, there were three workers: one responsible with detecting dead threads, another with deadlocks and the last with infinite loops. Due to the fact that deadlocks and infinite loop detection is performed using *PerfMon*, we combined last two threads in a single one.

The thread responsible with detecting dead threads uses IPC mechanism provided by the microkernel. Inter-process communication (IPC) is the central component of the VMXL4 system design. VMXL4 facilitates both communication between threads residing in different address spaces and threads within the same address space. The IPC system call provides the means for the exchange of short messages between two threads in the system. Using IPC call error code, the worker thread can find out if the monitored thread is still alive. If the monitored thread died, an error will be sent announcing that the destination doesn't exist.

The second worker is responsible with detecting deadlocks and infinite loops using *PerfMon*. We choose three events to monitor:

1) Instruction fetch that causes a refill at the lowest level of instruction or unified cache. Each instruction fetch from normal cacheable memory that causes a refill from outside of the cache is counted. Accesses that do not cause a new cache refill, but are satisfied from refilling data of a previous miss are not counted. This counter increments for speculative instruction fetches and for fetches of instructions that reach execution.
2) Immediate branch architecturally executed, taken or not taken. This counter counts for all immediate branch instructions that are architecturally executed, including conditional instructions that fail their condition codes.
3) Software change of PC (Program Counter), except by an exception, architecturally executed. This counter only increments for instructions that are unconditional or that pass their condition codes.

Detecting deadlocks is made by interrogating the software change of PC. A thread in a deadlock is blocked in an IPC call so no instructions are executed. If PC remains unchanged for a period of time (defined by the programmer), the worker considers that monitored thread is in a deadlock and notifies the monitor thread.

Infinite loop detection is more complex. A loop is a sequence of instructions that is continually repeated until a certain condition is reached. In order to detect an infinite loop, the assigned worker must analyze the monitored thread execution for a certain period of time using the below recursive formula:

- $P_n = \alpha_n * P_{n-1} + (1 - \alpha_n) * I_n, P_0 = 0$
- $\alpha_n = \beta * \alpha_{n-1} + (1 - \beta) * I_n, \alpha_0 = 0, \quad \beta$ a very small value (e.g. 0.05).

$P > L$ for the last T steps, $P_n, \alpha_n, I_n, \beta, L \in [0, 1]$, where P_n is the general prediction infinite loop factor at moment *n* based on all the infinite loop prediction up to moment *n*, I_n is the instant prediction infinite loop factor at moment *n* based only on the thread's state at moment *n*, state resulted from the values obtained from the hardware registers. A thread can be considered to be in an infinite loop, if its associated general prediction infinite loop *P* value is a higher or equal to threshold *L* for the last *T* steps.

B. Intersecure Domain High Availability

In order to ensure high availability if the local monitor from one secure domain crashes, we created a special secure domain which runs a global monitor. As you have seen in the architecture, we call this special secure domain the SecDom0. The monitor running here is responsible for monitoring the root task from each secure domain. There is no reason to monitor all threads in each secure domain because if the root task, the thread which is responsible with resource allocation, fails, the entire secure domain is compromised.

The detection is done by using PerfMon syscall to interrogate the different root task counters. Usually when the system running in a secure domain, panics, it enters an infinite loop and no action is taken (for example in linux is executed a *NOP* opperation. We can use the loop detection mechanisms implemented in the intrasecure domain to detect the panic of the system and then restart that secure domain. As we mentioned before, checkpointing an entire secure domain would take too much time and resources. This would be unacceptable for a real time system.

SecDom0 and any another secure domain are logically separated, doesn't have access to each other resources. How can we do PerfMon? The VMXL4 microkernel is based on capabilities for resource access. We used this mechanism to make connection between secure domains and implemented a new capability for PerfMon syscall. Therefore, a root thread from one secure domain can make PerfMon syscall on another secure domain root thread's if it has the PerfMon capability. This capability is given at build time. Based on what capabilities the monitor has, it will determine what secure domains to monitor.

VI. Discussion

In the current implementation, HA library provides the possibility to start and stop the HA module, to monitor or unmonitor threads. HA module detects dead threads and threads in a deadlock.

Microkernel provides fast means for detection: IPC mechanism and PerfMon. The main problem is that the microkernel provides only mechanisms, not services. For testing purposes, we had to implement minimal requirements for memory management, address space management and thread management. For example, we had to maintain a list with all allocated thread

identification numbers (called ids), for that secure domain, in userspace. In order to restart a thread, the monitor needs the thread id. To continue with the example, we will describe a little of the microkernel internals. The microkernel, assigns two ids for a thread in the system. One is used at the secure domain level for management purposes (create thread, kill thread, etc.) and one is used at the microkernel level, for reply IPC calls (when the pager needs to announce the thread that have made a page-fault and is waiting for that mapping). If the mapping couldn't be done, it would appear a segmentation fault and, in consequence, the thread that generated the page fault, needed to be restarted. In order to accomplish this, we needed the thread id (secure domain level) to be able to free resources (kill the thread) and to recreate it. But at the pager level, where we detected the segmentation fault, we had only the global id. To solve this problem we had to maintain an association between global id and secure domain level id.

The current implementation starts only the faulty thread because we can't test the restart of an address space. We can't create a new address space with its own mapped physical segments (not sharing physicals segments with root space). This problem is a consequence of the fact that microkernel provides mechanisms and not services.

We had difficulties when using sleep functions, timer functions and read write lock mechanism. All of them used the hardware counter. Because it is used simultaneous by lots of applications, we used a synchronization server that runs in a separated secure domain to control access to the hardware register.

PerfMon brings an important limitation: there are only four hardware registers so we can monitor maximum four events simultaneously. For infinite loop detection, we use the number of cache refills, the number of immediate branch architecturally executed and PC (program counter) changes (this register is used by deadlock detection, too). We need more events to have a better detection such as branch misprediction, number of taken branches that are executed, predicted branches that are predicted taken, etc. Another useful event, that is not implemented in hardware would have been the number of backward jumps. This value and the number of branches would have been necessary to determine more accurately if a program is in an infinite loop.

For intersecure domain architecture, we managed to detect a panic in the system running on a secure domain. The next step is restarting the secure domain. This involves freeing up the resources (allocated memory, delete created threads), reseting hardware registers (e.g. IP, SP, etc). Due to the microkernel architecture, the monitor is running in user-space and it isn't able to access the microkernel memory for resource management. We have to find some workaroung to do this, without adding new system calls.

VII. Conclusion and Further Work

We have designed and implemented a library with methods to start and stop HA module, to register and unregister a thread for been monitored. The workers detect dead threads and those in deadlock (infinite loop detection is in progress). As future work, the HA library, will provide methods to start only some types of monitoring, for example the case when the programmer doesn't want to start infinite loop monitoring because the detection is expensive and it isn't necessary.

Continuing with intrasecure domain high availability, we have to restart an address space when something went wrong with one of threads. The current implementation starts only the faulty thread because we can't test the restart of an address space. We can't create a new address space with its own mapped physical segments (not sharing physicals segments with root space). As a further work, it's necessary to implement a system to track physical segments associated with root space in order to duplicate and map them in the new address space.

About fault detection, more tests must be performed in order to determine the values of parameters of the equations when we can tell with big precision that we are dealing with an infinite loop. There are two actions that can be done when something went wrong in the system: restarting the address space/secure domain or restoring the address space. In the future, HA library will provide methods to save the state of an address space when the programmer wants, such that, if a fault occurs, that address space will be restored to a stable state which was indicated by the programmer (e.g. a cpu intensive program such as calculating PI decimals, avoiding loose all work that has been done till the restore point).

In the intersecure domains high availability we manage to detect panic errors based on infinite loop detection. To use this module, we had implemented special capability for PerfMon syscall. As future work, we have to implement the restarting of a paniced secure domain. As we described in the previous section, this implies freeing resources, clearing the registers and start execution from the begining.

Acknowledgment

The authors would like to thank their coordinators dr. eng. Razvan Deaconescu and Gary Gibson from Virtual Metrix, for their support and dedication.

References

[1] Qnx. http://www.qnx.com/.

[2] A. Podgurski, D. Leon, P. Francis, W. Masri, M. M. Sun and B. Wang. Automated support for classifying sw failure reports. In *ICSE Conference Proceedings*, pages 465–474, 2003.

[3] Burcu Ozcelik, Kubra Kalkan and Cemal Yilmaz. An Approach for Classifying Program Failures. In *Second International Conference on Advances in System Testing and Validation Lifecycle*, 2010.

[4] H. Agrawal, J. Horgan, S. London and W. Wong. Fault localization using execution slices and dataflow tests. In *ISSRE Conference Proceedings*, 1995.

[5] J. F. Bowring, J. M. Rehg and M. J. Harrold. Active learning for automatic classification of software behavior. In *ISSTA Proceedings*, pages 195–205, 2004.

[6] M. Haran, A. Karr, A. Orso, A. Porter and A. Sanil. Applying classification techniques to remotely-collected program execution data. In *SIGSOFT Softw. Eng. Notes*, pages 146–155, 2005.

[7] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox and E. Brewer. Pinpoint. Problem determination in large, dynamic internet services. In *International Conference on Dependable Systems and Networks*, pages 0:595–604, 2002.

[8] QNX. *QNX Software Development Platform. High Availability Framework. Developer's Guide*.

[9] R. Santelices, J. A. Jones, Y. Yu and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *ICSE Conference Proceedings*, pages 56–66, 2009.

[10] W. Dickinson, D. Leon and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *ICSE Conference Proceedings*, pages 339–348, 2001.

[11] W. Dickinson, D. Leon and A. Podgurski. Pursuig failure: the distribution of program failures in a profile space. In *FSE Conference Proceedings*, pages 246–255, 2001.

[12] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE Conference Proceedings*, pages 480–490, 2004.