An OS-level Framework for Providing Application-Aware Reliability

Long Wang, Zbigniew Kalbarczyk, Weining Gu, Ravishankar K. Iyer
Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign, IL 61801
{longwang, kalbar, wngu, iyer}@crhc.uiuc.edu

Abstract

The paper describes the reliability microkernel framework (RMK), a loadable kernel module for providing application-aware reliability and dynamically configuring reliability mechanisms installed in RMK. The RMK prototype is implemented in Linux and supports detection of application/OS failures and transparent application checkpointing. Experiment results show that the OS hang detection, which exploits characteristics of application and system behavior, can achieve high coverage (100% in our experiments) and low false positive rate. Moreover, the performance overhead is negligible because instruction counting is performed in hardware.

1. Introduction

Operating systems enable collecting and extracting rich information on application execution characteristics (e.g., counts of executed instructions and program counter traces). This information can be exploited to design highly efficient application-aware reliability mechanisms, which are transparent to applications. While some of the existing operating systems may provide (by design) reliability support, e.g., IBM AIX [1], High-Availability Linux [2], the emphasis is on achieving system reliability rather than exploiting execution characteristics of applications to improve their reliability.

This paper describes the design, implementation, and demonstration of a reliability microkernel (RMK) architecture deployed as a loadable Linux kernel module and supporting real world applications (e.g., Apache web server). The RMK provides an infrastructure that enables the design and deployment of software modules for providing application aware reliability services. Examples include OS/application hang/crash detection and application transparent checkpoint. The architecture exploits processor-level features (debug registers and monitoring facilities available in current generation of processors) and operating system-exported interfaces to define a set of basic services (called *RMK pins*, analogous to hardware pins by providing well-defined functionalities and inputs/outputs). The pins are employed to design application-specific mechanisms (referred to as *RMK modules*) for runtime system monitoring and low-latency error detection. The attributes of the currently implemented architecture allow:

- **Design and deployment of application-aware reliability techniques.** The RMK wraps (or abstracts out) the functionalities of OS and the underlying hardware in RMK pins. Using the interfaces and services exported by the pins, the designer of reliability modules can focus on development of application-specific techniques/algorithms without detailed knowledge on the specifics of the OS and the underlying hardware. Several techniques were implemented to demonstrate how the OS knowledge of application execution can be used to provide error detection and recovery, as summarized in Table 1 (the OS knowledge of application execution is highlighted in bold).

Table 1: Reliability Mechanisms in the RMK

Technique	Description
Application Hang Detection	**Number of instructions executed within a code block (e.g. a loop)** is counted; if the count goes beyond a preset scope, a hang is flagged.
Application Crash Recovery	**Delivery of the terminating signal from OS to a process** is intercepted and if the signal is not handled, the process is recovered from its checkpoint.
Transparent Application Checkpoint	**Original pages written during the checkpoint interval** are backed up; when the application fails (while the system is still operational), the original pages are restored.
OS Hang Detection	**The count of instructions executed between two consecutive context switches** is monitored; if the OS hangs, it does not schedule processes, and the instruction count goes beyond the preset scope.

- **On-demand configuration and customization of reliability techniques.** The RMK enables on-demand configuration of reliability support provided to applications. The RMK pins and RMK modules can be installed or removed on demand.
- **Portability to different platforms.** The two-tier architecture of the RMK decouples the platform-dependent (RMK pins) and platform-independent (RMK modules) components of reliability techniques, i.e., same set of RMK modules implemented on Linux can be directly ported (no code changes are required) to other operating systems as

12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)
0-7695-2724-8/06 $20.00 © 2006 IEEE

long as corresponding RMK pins are compiled for the target operating system.

Fault injection experiments conducted to evaluate the efficiency of the RMK implementation showed that the OS level mechanisms detected all application and OS hangs (due to injected errors) with very low false positive rate (1 out of 2000 experiments for application hang detection, and no false positive for OS hang detection). Additionally, the RMK-based mechanisms provide low-latency detection and transparent checkpointing with minimal impact on the system performance.

2. Related Work

There exists a rich volume of work that addresses issues of providing reliability services to applications using hardware and system support. Traditional microkernel systems like Mach [3] provide basic resource management and communications, and reliability architectures in IBM AIX [1], High-Availability Linux [2] are concerned with system reliability rather than exploiting application characteristics to improve application reliability. Hardware reliability frameworks such as Reliability and Security Engine (RSE) [4] provide application-aware mechanisms using programmable hardware modules to support detection/recovery of runtime errors.

Most existing techniques detecting hangs of OS and/or applications are timer-based and the timeouts are fixed values either preset or derived from profiling [5] [10]. They do not take into account the runtime execution of applications and the system. Therefore, the detection latency can be large. Moreover, approaches such as KHM [5] cannot operate when interrupts are disabled during OS hangs, while Linux NMI watchdog timer [10] does not detect hangs if the timer interrupt continues to function despite of the OS hang.

As there exist a large number of checkpoint techniques, we only list several of them here. Some checkpoint mechanisms preserve the entire process image, or even the whole system, for failure recovery or process migration [8] [7] [18], and others provide incremental checkpointing [6] [17] [16] [9]. Libckpt [6] checkpoints dirty pages of a process. Both [6] and [17] allow users to specify critical data for checkpointing and require application instrumentation. While an enhanced compiler can be used to automate the application instrumentation, user-provided directives are still needed to identify location (in the application code) where the instrumentation should be added. The cache-based checkpoint [16] needs additional hardware to enable storing application relevant state. More importantly, none of these checkpointing schemes uses OS-level knowledge to avoid inconsistency between application process image and the corresponding system state (e.g., for a checkpoint taken when the target application has pending I/O operations may be inconsistent with the I/O status at the time of recovery upon a failure.) Our checkpointing scheme handles this inconsistency.

3. RMK Framework

The RMK framework is developed as a loadable kernel module in Linux. The RMK has a two-level hierarchy (as shown in Figure 1): the lower level (RMK pins) interfaces with the system and hardware, while the upper level (RMK modules) hosts application-specific detection and recovery techniques. RMK pins encapsulate low-level services of the system and of the underlying hardware, and expose pin interfaces, which can be selectively used to build RMK modules. Each RMK module implements a specific error detection or recovery mechanism, e.g., crash and hang detection of applications, hang detection of the operating system, and transparent application checkpoint and recovery. The RMK core manages the installation and uninstallation of pins and modules; it also invokes pin functionalities on behalf of RMK modules. A set of RMK APIs is provided by the RMK core for management purpose.

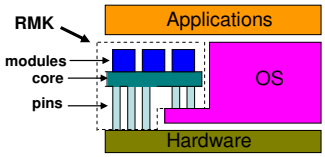


Figure 1: Reliability Microkernel in the system

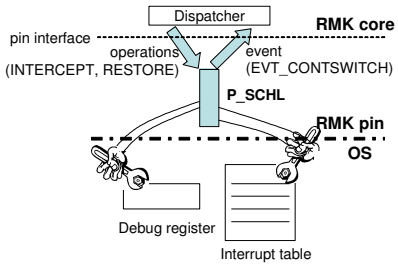
The abstraction introduced by two-level RMK structure enables: (i) use of standard operating system functions to create services essential in designing and implementing reliability mechanisms (encapsulated as RMK modules); (ii) portability of RMK modules across different operating systems, since pins implemented on different platforms export the same interfaces there is no need to modify the module's code; and (iii) transparent coordination between multiple modules and pins to avoid potential conflicts. For example, two implemented RMK modules, OS hang detection and application hang detection both intercept process context switches to detect hangs. Consider the situation when the application hang detection is installed while the OS hang detection module has already been in place. In this scenario, the installation of the application hang detection has the potential to overwrite the OS-level context-switch interception hook and inhibit the OS hang detection from working. The RMK coordination mechanism handles such situations and allows the two modules to function.

3.1. System-level RMK Interface

The RMK interfaces with the operating system (or more generically with the runtime environment) via RMK pins. A pin uses a collection of OS functions to construct a specific service essential in providing reliability mechanisms. In this sense, a pin implementation is platform specific, i.e., while the pin implementation on different platforms may be

different, the pin functionality and exported interface are to be the same regardless the platform[1].

As an example, Figure 2 depicts architecture (Figure 2a) and implementation (pseudo code in Figure 2b) of the *P_SCHL* pin, to intercept process context switch, which is an essential service in enabling application and/or OS hang detection. In modern operating systems context switches are transparent to applications, i.e., an application is not notified when a context switch happens. A common method to intercept context switches would be to instrument the scheduler. The P_SCHL pin, however, takes advantage of the debug exception mechanism to intercept context switches without any instrumentation of the operating system source code.



(a) architecture

```
int p_schl_init(void) {
   …
   rmk_register_pin(P_SCHL, p_schl_oper);
   …
}

int p_schl_oper(int operation_ID, void* arg) {
   get the interrupt vector# from arg;
   switch (operation_ID) {
      case INTERCEPT: // intercept context switches
         preserve the original debug exception handler;
         set the debug exception handler to my_DB_exp();
         set one debug register (DR3) to the scheduler function (schedule());
         enable DR3 to raise debug exception;
         break;
      case RESTORE: // restore the original setting
         disable DR3 to raise debug exception;
         clear DR3;
         restore the original debug exception handler;
         break;
   }
   …
}
```

```
int my_DB_exp(void)
{
   rmk_raise_event(P_SCHL, EVT_CONTSWITCH);
   call original debug exception handler;
   // the original handler disables debug exception
   set DR3 to scheduler();
   enable DR3 to raise debug exception;
}
```

(b) implementation code

Figure 2: The implementation of the P_SCHL pin in Linux

Generically, the interface exported by a pin consists of:

- a set of operations the pin can perform. In the case of P_SCHL the operations include: INTERCEPT and RESTORE (see Figure 2).
- a set of events the pin can produce given a trigger condition (a process context switch in the case of the P_SCHL pin) has occurred. In the case of P_SCHL the event set includes: EVT_CONTSWITCH (see Figure 2).

The INTERCEPT operation within the P_SCHL pin performs two primary tasks: (i) writes the scheduler entry point (the address of the first instruction of the *schedule()*) into one of the processor breakpoint registers (DR3 in the prototype implementation), thus forcing the processor to raise debug exception whenever *schedule()* is invoked[2] and (ii) modifies the system's interrupt vector table so that the *debug exception interrupt vector* points to the custom exception handler, *my_DB_exp()*, which (in addition to the default debug exception handler) generates an event, EVT_CONTSWITCH, to indicate context switch.

Table 2: RMK Pins in the RMK prototype

RMK Pin	Hardware/OS features [13]	Pin functionalities
P_PMC	Hardware counters available in modern processors for monitoring/measurement	Configures the counters; starts/stops counting; reads/writes counters; etc.
P_APIC	Advanced Programmable Interrupt Controller (APIC) provided in modern processors	Configures APIC for custom interrupt generation, e.g., generating an NMI interrupt when a performance counter reaches zero.
P_DBR	Debug facilities available in hardware (e.g. debug registers)	Sets up a debug exception at a particular location; generates an event upon debug exception
P_INTR	Interrupt handling in operating system.	Generates an event upon an interrupt through installed hooks
P_SCHL	Contexts are switched by the process scheduler in operating system.	Intercepts context switches; generates an event upon a context switch.

A small set of RMK APIs (*use_pin (), release_pin (), issue_cmd (), subscr_event (), and unsubscr_event()*) is implemented to enable transparent subscription to events and invocation of pin operations by RMK modules. In our example, OS and/or application hang detection module subscribes to the event exported by the P_SCHL pin. The subscription table (i.e., mapping between an event and a module or modules), is maintain by the dispatcher (see Figure 2a) and populated at the time of the module(s) initialization.

Although RMK pins deal with system specifics, no kernel source patch or recompilation is needed for pin implementation and deployment because the interfaces exported by operating systems, and debugging and monitoring facilities available in modern processors can be exploited for this purpose. For example, Linux exports a large number of kernel functions and variables in a symbol list stored in the file /boot/System.map (23306 symbols are exported in Linux 2.6.11.3). Table 2 lists examples of RMK pins currently implemented in the RMK prototype.

3.2 Application-level RMK Interface

Applications are monitored by RMK modules, which implement application-specific detection and recovery techniques. While most modules are application transparent, for some RMK modules application may need to be instrumented for example by using an enhanced compiler, which can insert a system call in the application code to enable invoking a given RMK module.

An RMK module can protect a set of applications on the system. RMK allows users to associate set of applications with RMK module(s). This information is kept in the dispatcher of RMK core. Moreover, RMK modules can be installed and removed on demand by users. A specially

[1] It is assumed that the basic functionalities (e.g., scheduler) are available across the operating systems considered here.

[2] In Linux *schedule()* is always invoked when a context switch occurs.

designed RMK module, Configuration Manager, enables RMK reconfiguration.

3.3 RMK Core

The RMK core is located between RMK modules and RMK pins, and is responsible for maintaining mapping between events generated by pins and modules subscribing to the events, delivering the events to modules, dispatching operation requests to pins and management (e.g., installation of a pin), and configuring pins and modules. The RMK core consists of four components, illustrated in Figure 3:

- ***pin manager,*** which is responsible for registration and loading/unloading of RMK pins;
- ***module manager,*** which handles installation and removal of RMK modules;
- ***dispatcher,*** which maintains subscription information for events generated by pins (i.e., mapping between events and RMK modules) and delivers the events to modules according to the subscription list; and
- ***RMK communication channel,*** which facilitates remote invocation of pins in a networked environment, i.e., enables requesting a pin operation or subscribing to an event generated by a pin on a remote node. This remote pin invocation facilitates design of distributed reliability mechanisms in RMK.

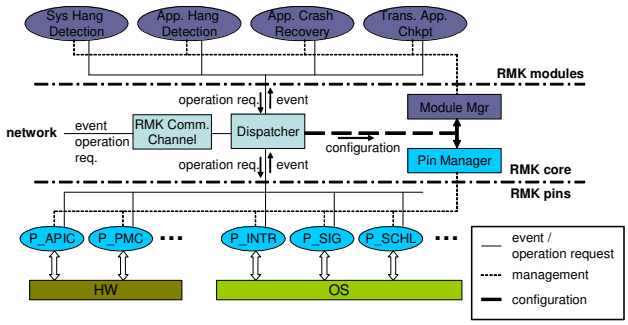


Figure 3: RMK Architecture

On-demand Configuration of RMK. RMK modules and pins can be deployed and removed on demand. When a module is installed, it acquires services from a set of pins. If required pins are not available in the RMK (e.g., specific pins for a new module), they are automatically loaded into memory from a permanent storage, e.g., a disk. The on-demand RMK configuration is initiated by the dispatcher and carried out by the pin/module managers (see Figure 3).

We now provide an illustration of how the RMK is configured to meet the needs of the specific error detection mechanisms. Assume that the RMK has installed only one module, *the OS hang detection module,* which requires five pins (P_SCHL, P_PMC, P_APIC, P_DBR, P_INTR) to be loaded into memory. A new module, the *application hang detection module (AHD)*, is to be deployed to protect an application. The AHD module uses six pins: P_SCHL, P_PMC, P_DBR, P_INTR, P_PERI, P_SYSC (the complete list of existing RMK pins are in the technique report [19]). Now, during initialization of the AHD module, services from these six pins are requested via RMK APIs. On receiving the service requests, the dispatcher forwards them to the pin manager, which finds that P_PERI and P_SYSC are not loaded into memory. The pin manager then loads the two additional pins from the disk, and sends a success response to the module via the dispatcher. If the required pins are not found, an error response is reported. After the successful module initialization, the dispatcher configures the event subscription table to establish corrected mapping between the new deployed AHD module and the events generated by the pins.

RMK Self-checking. It is possible that errors in the RMK may cause system failures. The dispatcher, pin/module managers and communication channels are well implemented and thoroughly tested. Due to the simplicity of RMK pins, errors in pins implementations can be considered negligible. Errors in RMK modules are confined using the following method. Whenever an operation of a module is invoked, RMK records the module ID. The record is cleared after the operation is finished. As there is no recursive invocation of module operations, the recorded ID always indicates currently executing module. When an error in a module triggers a kernel exception, the RMK intercepts the exception through the P_INTR pin, checks the recorded module ID, and unloads the module (the module can be reloaded immediately after the unloading if it is preferred).

The next sections discuss research challenges faced while developing the RMK. Several modules are used as examples in the discussion. An in-depth discussion on implementation issues is provided in [19].

4. System Hang Detection Module (SHD)

The operating system is subject to hangs due to, e.g., poorly-written drivers with blocking operations[3] and unreleased locks/mutexes. Chou et al. [12] claim that, 34% of kernel bugs in Linux 2.4.1 potentially lead to system hangs. The System Hang Detection module (SHD) provides low-latency detection and recovery of OS hangs.[4]

The underlying fault model for an OS hang is the following: an operating system in a hang state will not relinquish the processor and hence, it will not schedule any processes[5]. Based on this fault model OS hang detection can be constructed as follows:

1. Use an OS-level counter to keep track of instructions executed by the application processes and the OS

[3] For example, a device driver performs a blocking operation to acquire a mutex which is held by another process. As the blocking mutex-acquisition operation is performed in the kernel, the OS hangs.

[4] This section focuses on OS hangs with instructions being executed. For OS hangs with the processor halted, an augmented version of SHD is developed (see [19] for details).

[5] An external hardware watchdog could be used to force system reboot whenever the watchdog is not reset within a predefined timeout interval. In this case, however, one cannot distinguish whether the OS has hung or simply the process, designated to periodically reset the watchdog, has crashed/hung. In both cases the system reboot is initiated.

(system calls, device drivers, etc.) on behalf of the applications.
2. Update the system progress periodically (e.g. on each context switch).
3. If there is no progress update (e.g. no context switch) in the system, it is a clear indication of OS hang.

Important issues are to: (i) determine a time period at which to measure the counter and perform the update, (ii) have a mechanism to continuously and accurately measure the number of instructions being executed in the selected time period, and (iii) raise an alarm without additional hardware when an OS hang is present.

In a system executing a set of same-priority tasks, the count of instructions executed between two consecutive context switches (*continuous-execution-instruction-count*) is a finite number. If the OS hangs, it does not schedule processes. As a result, the instruction count grows beyond a limit. In principle this is determined by the time slice allocated by the OS for a process between two consecutive context switches (100ms for normal processes in Linux 2.6), and the count of executed instructions within the time slice has a bound (calculated as a product of the CPU speed and the time slice). However, in most of cases this bound is a large number, since it is common that a process voluntarily yields the CPU when it is waiting on resources (e.g., asynchronous I/O input), or it is idling for a certain time period (e.g., sleep functions or blocking synchronous I/O operations). In order to provide the low-latency OS hang detection, a much lower bound can be measured for the continuous-execution-instruction-count. A history of instruction counts collected over several time slices is employed to guide the estimation of the low bound.

The SHD module implements OS hang detection using two counters:

(i) a *profiling counter* – keeps track of the number of instructions executed in the current time slice by a process.

(ii) a *checking counter* – maintains a "*check value*", which is the running maximum of count of instructions executed in a time slice and obtained over a sliding window of approximately 50 time-slices.

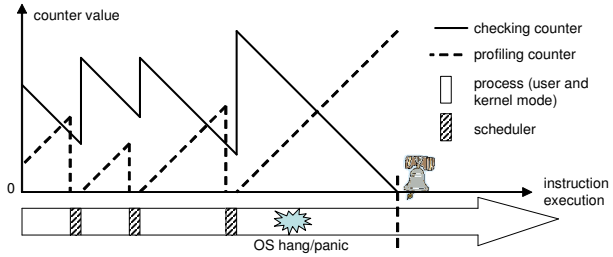


Figure 4: OS Hang Detection using instruction counting

Figure 4 illustrates the principle the SHD applies for OS hang detection. The horizontal axis represents instruction stream execution on the processor. The dashed and solid lines in the figure illustrate the evolution of the values in the two counters as a function of instruction execution. The profiling counter (the dashed line) increments with instruction execution count for each process during a time slice. When a context switch occurs the counter value is recorded by the SHD, and the counter is reset to zero to prepare for counting the instructions for the next scheduled process.

The checking counter (the solid line) detects OS hangs. It is set to an initial value, check value, after each context switch, and decrements with instruction execution. The check value is estimated according to the profiled count history (as discussed above). One can see from Figure 4 that when a higher value is recorded by the profiling counter, the check value increases. The instruction counts recorded over a reasonable number (e.g., 50) of recent time slices are used to compute the check value, which is to be loaded into the checking counter.

Recall that the fault model for the OS hang detection assumes that the OS continues to execute instructions without relinquishing control to application processes, i.e., the context switch is no longer invoked. Thus, in this situation the checking counter is not reset and, finally, it reaches zero (indicated as a bell alarm in Figure 4). At that time the counter raises a performance counter overflow interrupt (PMI) to the APIC, which then issues a Non-Maskable Interrupt (NMI) to the processor. The NMI handler initiates system reboot. The profiling-based checking makes the detection low-latency while adapting to changes in the system execution.

Check value determination. The check value is naturally bounded as the product of the processor speed and the time-slice, and a tighter bound may be achieved with knowledge of the instruction counts in a number of recent time slices. However, if the system transitions from an idle state to a busy one the determined check value may be very small, causing a false alarm. In these cases, a default check value is applied (the product of the processor speed and half of the time slice in the current prototype). Note that a larger default value reduces false alarms but increases hang detection latency. Profiling can be applied to obtain an accurate initial check value.

Priority-based scheduling. We assume that in a balanced system tasks have the same priority. Although Linux supports priority-based scheduling, lots of practical systems only deal with equal-priority tasks. In these systems kernel operations (system calls, interrupt/exception handlers, device drivers) are completed within the time far less than the process timeslice appropriately selected during the system design. Therefore SHD is able to detect system hangs successfully. For systems with tasks of different priorities, the check values can be set by profiling instruction counts executed in the time slices of the high-priority tasks.

5. Application Hang Detection Module (AHD)

Applications (or individual application threads) may hang due to hardware faults, deadlock, failed I/Os, etc. The Application Hang Detection module (AHD) transparently

detects application hangs[6]. AHD exploits a fact that the count of instructions executed in a well-defined code block, or code section, is normally bounded in a fixed range [11][7]. If the instruction count goes outside the range, the application hang is flagged.

A code section is a block of code with a single entry. A code section is monitored as a unit by the AHD. Examples of code sections include: loop, loop body (i.e. one iteration of a loop), function, and mutex block (a code block between mutex acquisition and release). After a thread enters a code section, it must leave the section before it enters the section again. A recursively called function is not monitored as a code section. A code section may include multiple nested sections like nested loops, and code sections may overlap with other sections, e.g. a loop overlapping with a mutex block. Each code section has a unique ID within a process. Multiple threads may execute the same code section (the same piece of code) simultaneously.

The AHD keeps a logical counter for every thread running in a monitored code section. Consequently there is an array of counters to monitor a multithreaded application. When a thread enters a code section the corresponding counter starts counting instructions from zero (the counter is called *active*); when it leaves the section the counter stops counting (*inactive*). As the thread may be executing an instruction within multiple code sections, multiple counters for the thread may be active at the same time. Granularity of code sections can be selected to limit the number of sections to avoid large overhead (typically 3-6 code sections are monitored for an application).

Though the AHD uses an array of logical counters for application hang detection, actually only one hardware performance monitor counter is used for counting. By using the hardware counter and the runtime system monitoring, the AHD detects application hangs with a low overhead and a low latency. The check value for a code section is derived from the execution history of the section, similar to the check value determination in SHD (initial check values are obtained from profiling).

6. Transparent Application Checkpoint Module (TAC)

Existing checkpointing schemes for individual applications usually take a snapshot of the application process image. However, as an application may have issued I/O operations which are pending in the system and not finished by the checkpointing time, the snapshot of the process image, if restored upon a failure, may be inconsistent with the I/O status in the system. File reading/writing and network communication are examples of the I/O operations. Previous application checkpointing scheme, e.g. Plank's libckpt [6], relies on user-level knowledge to decide when to take checkpoint for addressing consistency with I/O state, OS-issued signals and message queues. This method is not always effective as we cannot expect every program developer has enough system-level expertise to handle the consistency issue.

The TAC applies an approach similar to libckpt to incrementally checkpoint the dirty memory pages. However, doing so TAC: (i) exploits the OS-level knowledge (collected by RMK pins) on the application execution and (ii) decides (transparently to the application) when and how to checkpoint so to avoid any inconsistency between the application image and the system state. In addition, synchronization between threads in a multi-thread process and consistency of process checkpoints in a distributed application can also be addressed transparently with the OS-level knowledge (e.g., messages can be logged by invoking operations of the corresponding pin).

This section discusses how TAC addresses the consistency issue in a generic checkpointing method [8]. We implemented the TAC checkpointing on Linux-based platform augmented with RMK framework and evaluated the performance of the proposed solution by conducting experiments with real-world application.

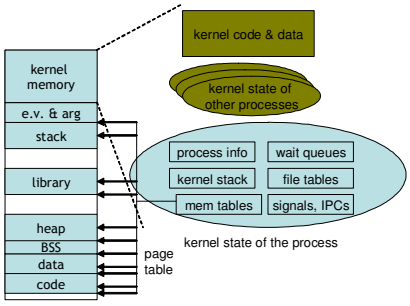


Figure 5: The user/kernel state a process in Linux

Figure 5 illustrates a breakdown of the address space of a process execution on top of Linux. The kernel state of the process includes the process info, kernel stack, memory tables, signal/IPC status, wait queues and tables for opened files. The process info (in Figure 5) contains process attributes (process ID, schedule priority), process relations (parent, child), and user information (uid, gid). The wait queues deal with the pending asynchronous I/O operations and process/thread synchronization. The user state of the process consists of memory pages constituting the segments of code, data, heap, stack and dynamically loaded libraries.

The checkpoint taken by the TAC module includes: (i) the modified user memory pages and (ii) the processor state. The TAC module eliminates the need to checkpoint the kernel state of the process by taking a checkpoint only

[6] The hung thread is either continuously scheduled on the processor, as in an infinite loop; or not scheduled at all. This section focuses on the former case. More discussion on the latter is in [19].

[7] Note that the problem of determining the worst case execution time, or instruction count, of a program or a code block is in general undecidable and is equivalent to the halting problem.

[8] When the characteristics of application behavior are utilized, the generic checkpointing scheme can be customized to specific applications to achieve a better performance. A custom checkpointing mechanism for Apache web server is discussed in [19].

when the kernel state is in a *safe point*, i.e., there are no pending I/O, signals, or IPC messages in the context of the application process. Only the opened files table (names and current seek offsets of all open files) and the memory table in the kernel state of the process are checkpointed. When the process is recovered upon a failure, the file table and memory table are restored in the process kernel state with no pending I/O, signals, or IPCs messages, and with an empty kernel stack. Consequently, after restoring the checkpointed user memory pages, the recovered process is in a consistent state. The steps given below outline steps in the TAC algorithm for application checkpointing:

1. *Periodically initiate a checkpoint by setting a chkpt_started flag to 1.*
2. *Check whether there is a pending I/O operation, signal, or IPC message in the target process. If TRUE go to sleep; otherwise, go to step 6.*
3. *Upon completion of an I/O operation (or signal processing or IPC message processing), check the chkpt_started flag (this is done using functionality provided by P_FILE/P_NET pins). If the flag is set, the pin raises an event EVT_IO_COMP/EVT_NET_COMP to notify the TAC module.*
4. *Upon initiation of an I/O operation by the process (the P_FILE/P_NET pins), check the chkpt_started flag. If the flag is set, postpone the I/O operation. New IPC messages (check by the P_IPC pin) are handled in similar fashion as I/O operations. New arriving signals (check by the P_SIG pin) are allowed to be immediately processed.*
5. *Upon notification of new event EVT_*_COMP, go to step 2.*
6. *Checkpoint the application process (there are no pending I/O, signals, or IPCs): (i) set all the user memory pages of the process as read-only (using the P_MEM pin), (ii) save the names and current seek offsets of all open files, and (iii) preserve memory table and CPU state.*
7. *Clear the chkpt_started flag and wait for the duration of the checkpoint interval to initiate the next checkpoint.*
8. *Upon a write to a read-only user page in the process, handle a segmentation fault signal raised by the system (and captured by the P_SIG pin, which raises an event EVT_SIG_SEGV to notify the TAC; duplicate the page in backup memory (hosted by a user-level dummy process) and enable writes to the page (using the P_MEM pin).*

7. Experimental Evaluation

The RMK prototype is implemented as a loadable kernel module in Linux 2.6.11 on a Pentium 4 processor (1.5G Hz). The RMK pins are implemented as part of the RMK kernel module by taking advantage of the kernel functions and variables exported by Linux as well as the monitoring/debugging capability of the hardware. Experiments were conducted on the RMK prototype to evaluate its effectiveness and performance overhead. NFTAPE [14], a software toolset, was used to launch fault injections and assess effectiveness of individual RMK modules in terms of their error coverage. Due to space limitation, the description focuses only on the experiment results for the SHD module. The evaluations of the AHD and TAC modules, as well as the performance overhead of the RMK can be found in the technical report [19].

Experiment setup. In order to assess effectiveness of the OS hang detection we need to create operational conditions which are likely to lead to OS hangs. Towards this goal, a victim device driver (CRC-driver), which calculates cyclic redundancy code, is implemented and attached to the operating system. At runtime, faults are injected to the CRC-driver to induce OS hangs. A synthetic application invokes the driver to compute CRC on a selected data set and generate sufficient system load to enable activation of errors injected into the CRC-driver.

Outcome categories. Outcomes from error injection experiments are classified according to the categories given in Table 3.

Results. Table 4 gives distribution of fault injection outcomes. Strictly speaking, as errors propagate within the system, a single fault may trigger multiple failures and the first observed failure outcome is reported in Table 4.

Table 3: Outcome categories

Outcome category	Description
Correct output	The application produces correct output. The error may not be activated, or activated but not manifested; The OS performs normally during the experiment period (15s);
Fail silence violation	The application produces incorrect output; The OS performs normally during the experiment period (15s);
Kernel exception	The application terminates abnormally; The OS raises an exception: invalid kernel paging request, kernel NULL dereference, general protection, invalid operand or others;
Silent hang	The application does not complete; The OS hangs without reporting any exception;

Table 4 indicates that 9.0% of injected errors lead to OS hangs, and the OS hang detection provides the 100% coverage for the hangs. Table 5 gives five examples of error propagations leading to OS hangs. One can see that a propagated error may cause different kinds of kernel exceptions (Kernel NULL dereference and Invalid kernel paging request are caused in example 3). Furthermore, critical data structures in the operating system may be corrupted by invalid instruction execution (GDT, *Global Descriptor Table*, and TSS, *Task Stack Segment*, in the system are corrupted in example 4 and 5). Surprisingly, kernel exceptions may be triggered repeatedly for a number of times (*spin_lock_unreleased* is reported for more than 30 times before GDT is corrupted in example 5).

Table 4: Error injections in the victim driver (1346 experiments)

First observed failure outcome	Number of manifestations	Number of hangs	Number of detected hangs
Correct output	391 (29.0%)	1*	1
Fail silence violation	380 (28.2%)	0	0
Kernel exception: invalid kernel paging request	249 (18.5%)	13	13
Kernel exception: kernel NULL dereference	93 (6.9%)	6	6
Kernel exception: general protection	66 (4.9%)	0	0
Kernel exception: invalid operand	62 (4.6%)	0	0
Kernel exception: other	15 (1.1%)	11	11
Silent hang	90 (6.7%)	90	90
Total failed	**956**** **(71.0%)**	**121 (9.0%)**	**121**

*In this experiment the application produces correct results, however, the error propagates and causes an invalid kernel paging request, which further leads to the OS hang.
**The number includes the case marked with *.

SHD detects a hang after a predetermined number of instructions executed without a context switch, and then reboots the system. Therefore, it is unknown how long, after the hang detection, the system remains in unoperational state. Six experiments randomly selected from the 90 silent hangs are analyzed to determine what happened to the system after the hang detection. In two of the six experiments the system hangs for 2 seconds and then

reports an invalid kernel paging request; in another two the system hangs for 10+ seconds before another failure is reported; and in the remaining two experiments the system hangs for the time longer than the observation period (4 minutes). The analysis results show that without SHD the system undergoes a long period of invalid execution. It may finally trigger kernel exceptions or not trigger any exception but continue hanging. This study also indicates that, although the operating system provides checking for erroneous instruction executions (in 455 of 1346, or 33.8% of the experiments, the injected errors are detected by the system itself), the SHD of RMK provides a complement solution to detect erroneous instruction executions with a bounded latency.

Table 5: Examples of error propagations leading to OS hangs

Exa-mple	First observed outcome	Failure propagation trace
1	Invalid kernel paging request	Invalid kernel paging request -> Invalid kernel paging request -> Kernel panic -> hang detected
2	Invalid kernel paging request	Invalid kernel paging request -> Kernel NULL dereference -> hang detected
3	Kernel NULL dereference	Kernel NULL dereference -> Invalid kernel paging request -> Invalid kernel paging request -> Kernel panic -> hang detected
4	Other	gdt double fault -> tss double fault -> hang detected
5	Other	spin_lock unreleased -> spin_lock unreleased -> … -> spin_lock_unreleased -> gdt double fault -> tss double fault -> hang detected

8. Conclusions

The paper describes the reliability microkernel framework (RMK), a loadable kernel module for providing application-aware reliability and configuring reliability mechanisms installed in RMK. The RMK prototype is implemented in Linux 2.6.11 on a Pentium 4 processor and supports detection of application/system failures and transparent application checkpointing. The experimental evaluation of the RMK using real-world applications (Apache web server) shows that the proposed OS hang detection mechanism (SHD) can achieve high coverage (100% in our experiments) and low false positive rate (no false positive in our experiments). The performance overhead of SHD is negligible because instruction counting is performed in hardware and the context switch frequency is relatively low. Evaluation of coverage and performance overhead of the proposed application hang detection mechanism (AHD) and transparent application checkpoint mechanism (TAC) can be found in [19].

Acknowledgment

This work was supported in part by NSF grants CNS-0406351 (Next-Generation Software), CNS-05-24695, and ACI-0121658 ITR/AP, the Gigascale Systems Research Center (GSRC/MARCO), and Motorola Corporation as part of Motorola Center.

References

[1] "Open, secure, scalable, reliable UNIX for POWER5 servers", *AIX 5L for POWER Version 5.3,* IBM Corporation 2004.

[2] A. Robertson, "The Evolution of the Linux-HA Project", *UKUUG LISA/Winter Conference High-Availability and Reliability,* Bournemouth, UK, 2004.

[3] R. Rashid et al., "Mach: a foundation for open systems", in *Proc. 2nd workshop workstation operating system*. 1989.

[4] Nithin Nakka et al., "An Architectural Framework for Providing Reliability and Security Support". *The International Conference on Dependable Systems and Networks*, Florence, Italy, 2004.

[5] Daniel Beauregard, "Error injection-based failure profile of the IEEE 1394 Bus", MS thesis, University of Illinois Urbana Champagin, 2003.

[6] J. Plank et al., "Libckpt: Transparent checkpointing under Unix", Conference Proceedings, *Usenix Winter 1995 Technical Conference,* 1995.

[7] Steven Osman et al., "The design and implementation of Zap: a system for migrating computing environments", *ACM SIGOPS Operating Systems Review,* Volume 36, 2002.

[8] E. Pinheiro, "Truly-Transparent Checkpointing of Parallel Applications", internal report, Dec. 1997. http://www.research.rutgers.edu/~edpin/epckpt/epckpt.ps.gz

[9] Long Wang et al., "Checkpointing of Control Structures in Main Memory Database Systems", *The International Conference on Dependable Systems and Networks,* Florence, Italy, 2004.

[10] D. Bovet et al., "Understanding the Linux Kernel", 3rd Edition, Chapter 6.4.2 "Checking the NMI Watchdogs", *O'Reilly & Associates,* Inc., 2005.

[11] Y.-T. S. Li, et al., "Performance estimation of embedded software with instruction cache modeling", *ACM Trans. On Design Automation of Electronic Systems,* 4(3).

[12] A. Chou et al., "An empirical study of operating systems errors", *Symposium on Operating Systems Principles,* Banff, Canada, 2001.

[13] The IA-32 Intel® Architecture Software Developer's Manual, Intel Corp., 2006.

[14] D. Stott et al., "Dependability assessment in distributed systems with lightweight fault injectors in NFTAPE", *Proc. Of the Dependable Computing for Critical Applications Conf.*, 1998.

[15] S. Srinivasan et al., "Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging", *In USENIX Annual Technical Conference,* 2004.

[16] P. Bernstein, "Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing", *IEEE Computer,* Feb. 1988.

[17] Y. Huang et al., "Software implemented fault tolerance: Technologies and experience", In *Proc. IEEE Fault-Tolerant Computing Symposium,* 1993.

[18] L. Wang et al, "Modeling Coordinated Checkpointing for Large-Scale Supercomputers", *The International Conference on Dependable Systems and Networks,* Yokohama, Japan, 2005.

[19] L. Wang et al, "An OS-level Framework for Providing Application Aware Reliability", technical document, UILU-ENG06-2218, 2006.