An Operating-System-Level Framework for Providing Application-Aware Reliability

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

Abstract

Operating systems enable collecting and extracting rich information on application execution characteristics, including program counter traces, memory access patterns, and operating-system-generated signals. This information can be exploited to design highly efficient, application-aware reliability mechanisms that are transparent to applications. This 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 and application hang detection, which exploit characteristics of application and system behavior, can achieve 100% coverage and low false positive rates. Moreover, the performance overhead of RMK and the detection/checkpointing mechanisms is small (0.6% for application hang detection and 0.1% for transparent application checkpointing in the experiments).

1. Introduction

Operating systems enable collecting and extracting rich information on application execution characteristics, including counts of executed instructions, program counter traces, memory access patterns, and OS-generated signals. This information can be exploited to design highly efficient, application-aware reliability mechanisms that are transparent to applications. While some of the existing operating systems may provide (by design) reliability support, e.g., IBM AIX [3] and High-Availability Linux [4], the emphasis is on achieving system reliability rather than exploiting execution characteristics of applications to improve application 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 the current generation of processors) and OS-exported interfaces to define a set of basic services. These basic services are called *RMK pins*, which are analogous to hardware pins in providing clearly 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 the following:

- **Design and deployment of application-aware reliability techniques.** The RMK wraps (or abstracts out) the functionalities of the OS and the underlying hardware into RMK pins. Using the interfaces and services exported by the pins, designers of reliability modules can focus on developing application-specific techniques/algorithms without detailed knowledge of the specifics of the OS or the underlying hardware.

Several techniques have been implemented to demonstrate how OS knowledge of application execution (e.g., counts of instructions executed within a monitored code block, memory access patterns, or delivery of OS signals to applications) can be used to provide error detection and recovery. Table 1 briefly summarizes the techniques currently available in the RMK.

Table 1: Reliability Mechanisms in the RMK

Reliability Service	Application Execution Pattern	Technique Description
Application Hang Detection	Number of instructions executed within a well-defined code block, e.g., a loop	Number of instructions executed within the code block is counted. If the count goes beyond a preset scope, a hang is flagged.
Application Crash Recovery	OS signal delivered to application	Delivery of the terminating signal to a process is intercepted. If the signal is not handled, the process is recovered from its checkpoint.
Transparent Application Checkpoint	Memory access patterns	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	Number of instruction executed between two consecutive context switches	The count of instructions executed between two consecutive context switches is a finite number. 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. RMK pins and RMK modules can be installed or removed on demand.
- **Portability to Different Platforms.** The two-tier architecture of the RMK (pins and modules) decouples the platform-dependent and platform-independent components of reliability techniques. For example, the same set of RMK modules implemented on Linux can be directly ported (with no code changes required) to Solaris, FreeBSD, or Windows systems, as long as corresponding RMK pins (the platform-dependent components) are compiled for the target operating system.

Fault injection experiments conducted to evaluate the efficiency of the RMK implementation show that the OS-level mechanisms detected 100% of application and OS hangs (due to injected errors) with a very low false positive rate (1 out of 2000 experiments for application hang detection, and no false positives for OS hang detection). Additionally, the RMK-based mechanisms provide low-latency detection and transparent checkpointing with minimal impact on system performance (0.6% performance overhead for application hang detection and 0.1% overhead for transparent application checkpointing).

2. Related Work

Table 2 summarizes representative examples of studies (in academia and industry) and systems (commercial and research prototypes) that address issues of providing reliability services to applications using hardware and system support. Microkernel systems such as Mach [6] and Chorus [7] [26] provide basic resource management and communications. Reliability architectures in IBM AIX [3] and High-Availability Linux [4] are designed to be closely coupled with the systems. They are mostly concerned with system reliability rather than exploiting application characteristics to improve application reliability. Hardware reliability frameworks such as the Reliability and Security Engine (RSE) [9] provide application-aware mechanisms using programmable hardware modules to support the detection/recovery of runtime errors.

Table 2: List of related work

Category	Study/System	Reliability Features	Comments
Microkernel Architecture	Mach [6]	Reliability is not the primary focus.	Architectural basis to build other OSs.
	Chorus [7] [26][27]	Reconfigurable microkernel; applies event/exception handling for reliability; predicates are defined in wrappers of functional services to guard against incorrect state and error propagation.	Architectural basis to build other OSs. Wrappers of microkernel services used to check system health rather than support application-aware techniques.
Reliability Architecture	IBM AIX [3]	Virtual-server-based system protection; application hang detection; a daemon polls the OS kernel to check if processes are being starved.	OS-level support focused on system reliability.
	SGI IRIX	Process checkpointing dumps process image to disk.	Needs OS-level support for preserving the entire process image.
	High-Availability Linux [4]	Heartbeat used for detection of node failures; membership protocol used for group communication.	Needs support for system reliability, especially cluster health.
	Sentry [5]	Additional layer placed on top of OS provides error masking and system service guard.	Supports system reliability rather than application.
	ARMOR [12]	Self-checking middleware provides fault tolerance to applications.	Application instrumentation is required for implementing application-specific reliability techniques.
Hang Detection of OS or Applications	RSE [8][9]	Processor-level framework provides application-aware error detection, e.g., detection of OS and application hangs using hardware modules.	Reliability mechanisms are bound to hardware modules; needs OS support for application hang detection.
	Watchdog Device [10]	OS hang detection: an external PCI card is used as watchdog.	Extra hardware is required; long latency (timeout sometimes in minutes)
	KHM [11]	OS hang detection: a process periodically sets a mark, and the timer interrupt handler checks the mark to detect OS hangs.	Fails when interrupt is disabled; long latency when system is heavy loaded; large overhead.
	Linux NMI Watchdog [17]	OS hang detection: if there is no timer interrupt within a few seconds, the OS hang is detected.	Fails when the OS hangs with timer interrupts arriving and handled.
Application Checkpointing	Libckpt [13], LibFT [31]	Libraries invoked by applications dump application states and/or critical data.	Not application-transparent; user-level knowledge is required for high performance.
	Zap [14]	Virtual machine solution transparently checkpoints applications.	Checkpoints the entire process image.
	Epckpt [15]	New system calls added to the kernel provide transparent checkpoint.	Checkpoints the entire process image.
	Incremental Checkpointing [16]	Application-transparent incremental checkpoints are provided for main memory database	Application-used library is instrumented to support incremental checkpoints; custom solution for MMDB applications.
	Cache-based Checkpointing [29]	Checkpoint is triggered on cache misses; the checkpoint state is stored in memory and includes processor registers and cache.	Checkpoint interval is very short; error may propagate across checkpointing; large overhead; additional hardware.
	Coordinated Checkpointing [32] [33]	Checkpoint protocols are provided for distributed applications.	Checkpoints the entire process image; focuses on checkpoint protocol.
	Shadow Process Checkpointing [28]	Shadow process is forked to store the process image upon a checkpoint; copy-on-write is applied for dirty page copying.	Important process states like pid not preserved; inefficient copying of pages with only one/two writes; unnecessary allocation of process resources.
	Compiler-Assisted Checkpointing [34]	Compiler inserts checkpoint functions in programs according to user-provided directives for checkpointing critical data.	Not application-transparent; user-level directives are required.

Most existing techniques detecting hangs of the OS and/or applications are timer-based, and the timeouts are fixed values either preset or derived from profiling. They do not take into account the runtime execution of applications and the system. Therefore, the detection latency can be large. Some existing detection mechanisms cannot operate when the interrupts are disabled due to OS hangs, e.g., KHM [11]. Others, e.g., the Linux NMI watchdog timer [17], does not detect hangs if the timer interrupt continues to function despite of the OS hang.

As there are many checkpoint techniques, we do not list all of them in Table 2. Some checkpoint mechanisms preserve the entire process image for failure recovery or process migration [15] [14], while others provide incremental checkpointing [13] [31] [29] [28]. Libckpt [13] checkpoints dirty pages of a process. Both [13] and [31] allow users to specify critical data for checkpointing, and both require application instrumentation. While an enhanced compiler can be used to automate the application instrumentation, user-provided directives are still needed to identify the location (in the application code) where the instrumentation should be added [34]. Cache-based checkpointing [29] needs additional hardware to enable storing of application--relevant state. More importantly, none of these checkpointing schemes uses OS-level knowledge to avoid inconsistency between the application process image and the corresponding system state. For example, 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 failure. Our checkpointing scheme handles this inconsistency.

3. RMK Framework

The reliability microkernel (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 they 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, or transparent application checkpoint and recovery. The RMK core manages the installation and de-installation 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.

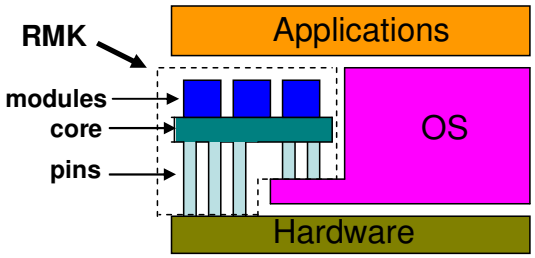
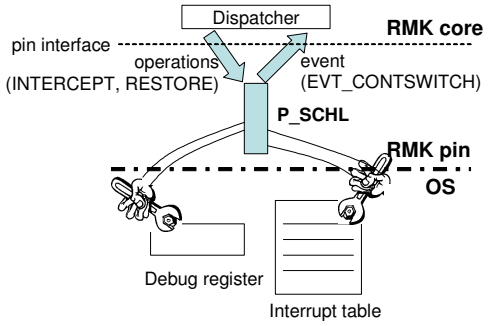


Figure 1: The RMK in the system

The abstraction introduced by the 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 operating systems, since pins implemented on different platforms export the same interfaces (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 in which application hang detection is installed while the OS hang detection module is already in place. In this scenario, the installation of application hang detection has the potential to overwrite the OS-level context-switch interception hook and inhibit 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 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 intended to be the same regardless the platform[1].



(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 an example pin, P_SCHL in Linux

Table 3: RMK pins in the RMK prototype

RMK Pin	Hardware/OS Features	Pin Functionalities
P_PMC	Hardware counters are available in modern processors, which permit monitoring and measuring processor performance parameters, including instruction count, TLB access, and cache reference.	Configures the counters to measure specific parameters; starts/stops counting; reads/writes counters; generates a PMI interrupt to APIC when a counter reaches zero.
P_APIC	Advanced Programmable Interrupt Controller (APIC) is provided in modern processors. It receives interrupts from processor pins or external interrupt controller and delivers interrupts to the processor core.	Configures APIC for custom interrupt generation, e.g., generating an NMI interrupt when a performance monitor counter raises a PMI to APIC.
P_DBR	Debug facilities are available in hardware, including debug exception and debug registers.	Sets up a debug exception at a particular location; installs debug exception handler; generates an event upon debug exception.
P_INTR	Interrupt handling is in the OS.	Installs hooks for specific interrupts; generates an event upon an interrupt.
P_SIG	Signals are delivered by the operating system to processes.	Intercepts particular signals to processes; generates an event upon a signal delivery.
P_MEM	Memory manipulation is provided in the OS.	Copies memory pages; sets page properties.
P_SCHL	Contexts are switched by the process scheduler in the OS.	Intercepts context switches; generates an event upon a context switch.
P_PERI	System-level periodic jobs are performed by the work queues of the OS.	Sets up a system-level periodic job at a specified interval; generates an event upon interval expiration.
P_NET	Network communications are performed by the system.	Intercepts network activities, including message sending and arrival; generates events upon these activities.
P_SYSC	A variety of system calls are provided by the system on behalf of applications.	Intercepts specific system calls; generates an event upon a system call.
P_FILE	Functions are exported to manage the table of opened files as well as to read/write to the files.	Wraps the manipulation of the file table; generates an event when a monitored file read/write completes.
P_IPC	The system provides IPC mechanisms, such as pipes, message queues, and semaphore.	Wraps the manipulation of IPCs; generates an event when a monitored IPC occurs.

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

As an example, Figure 2 depicts architecture (Figure 2a) and implementation (pseudocode in Figure 2b) of *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 occurs. 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.

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 P_SCHL pin). 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) It 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 a debug exception whenever schedule() is invoked[2]. (ii) It 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.

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 the invocation of pin operations by RMK modules. In our example, the OS and/or application hang detection module subscribes to the event exported by the P_SCHL pin. The subscription table (i.e., the 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 or deployment. This is because the interfaces exported by operating systems and the 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 3 lists the 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 the application may need to be instrumented, for example by using an enhanced compiler, to 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. The RMK allows users to associate a set of applications with RMK module(s). This information is kept in the dispatcher of the RMK core. Moreover, RMK modules can be installed and removed on demand by users. A specially designed RMK module, Configuration Manager, enables RMK reconfiguration.

---

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

3.3 RMK Core

The RMK core is located between RMK modules and RMK pins. It is responsible for maintaining mapping between events generated by pins and modules subscribed to the events, delivering the events to modules and 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.

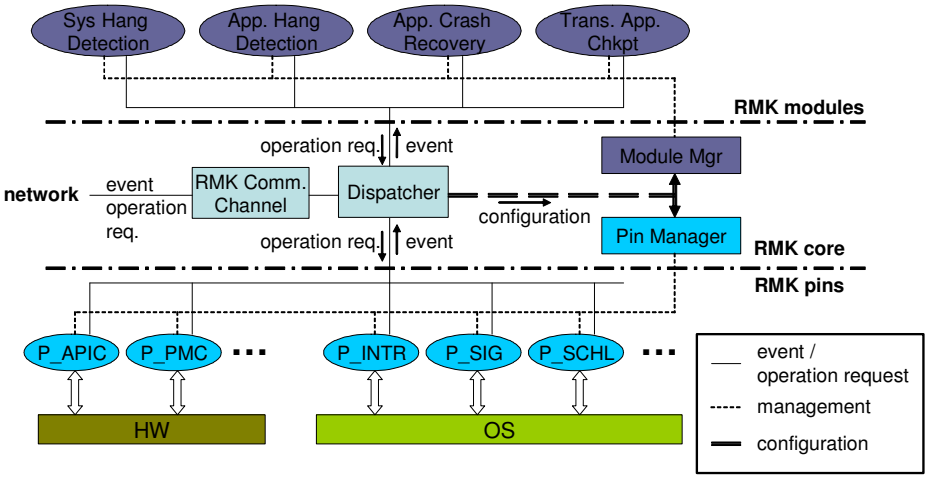


Figure 3: RMK architecture

On-Demand Configuration of the 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 the required pins are not available in the RMK (e.g., specific pins for a new module), they are automatically loaded into memory from 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. Now, during initialization of the AHD module, services from these six pins are requested via the 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 sent. After the successful

module initialization, the dispatcher configures the event-subscription table to establish corrected mapping between the newly deployed AHD module and the events generated by the pins.

RMK Self-Checking. 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 pin implementations can be considered negligible. Errors in RMK modules are confined using the following method. Whenever an operation of a module is invoked, the 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 the 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 that is preferred).

The next sections discuss research and implementation challenges faced while developing the RMK. Several modules are used as examples in the discussion.

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. [21] claim that 34% of kernel bugs in Linux 2.4.1 potentially lead to system hangs. The processor may be executing non-HLT instructions or is halted by executing a HLT instruction during an OS hang. The System Hang Detection module (SHD) provides low-latency detection of and recovery from OS hangs.

An external hardware watchdog can be used to force a system reboot whenever the watchdog is not reset within a predefined timeout interval. In this case, however, one cannot determine whether the OS has crashed/hung or the heartbeat process, designated to reset the watchdog periodically, has crashed/hung. In either situation, the system reboot is initiated, although in the latter case, a system reboot might be not necessary. Also, the detection latency might be significant, since the timeout interval for resetting the watchdog timer is usually a matter of seconds (to reduce false alarms and time overhead due to the heartbeat).

OS kernel instrumentation and watchdog timer modifications are required to reduce latency in detecting OS hangs and to avoid fault positives (i.e., unnecessary OS reboot in the case of a heartbeat process crash/hang). For example, one could instrument the OS scheduler to send a heartbeat message to the watchdog on every context switch. If no heartbeat arrives within a certain time interval (i.e., operating system does not schedule processes), an OS hang is declared.

The approach in this paper enables low-latency, transparent OS-hang/crash detection. The detection mechanism is designed and implemented as a light-weight kernel driver, and hence, it does not require instrumentation of the kernel code and is fully transparent to the system.

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. Based on this fault model, OS hang detection can be constructed as follows:

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

1. Use an OS-level counter to keep track of instructions executed by the application processes and the OS (system calls, device drivers, etc.) on behalf of the applications;
2. Update the system progress periodically (e.g., on each context switch); and
3. If there is no progress update (e.g. no context switch) in the system, this is a clear indication of an OS hang.

Important issues are: (i) to determine a time period during which to measure the counter and perform the update, (ii) to 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 (i) the time-slice allocated by the OS for a process between two consecutive context switches (100ms for normal processes in Linux 2.6) and (ii) the fact that 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 idling for a certain time period (e.g., sleep functions or blocking synchronous I/O operations). In order to provide 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 used to guide the estimation of the low bound.

The SHD module implements OS hang detection using two counters:

(i) ***Profiling counter,*** which keeps track of the number of instructions executed in the current time slice by a process, and

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

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 as instructions execute 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.

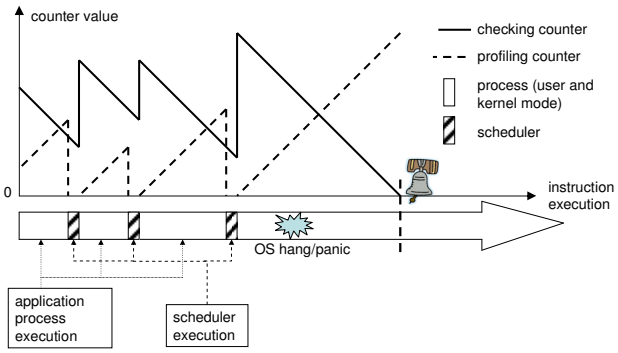


Figure 4: OS hang detection using instruction counting

The checking counter (the solid line) detects OS hangs. It is set to an initial value, the *check value*, after each context switch, and it 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 sliding window of 50 recent time-slices are used to compute the check value that 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 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 an NMI (Non-Maskable Interrupt) 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.

4.1 Issues

Hangs Due to a Halted Processor. Recall that according to our earlier definition, the OS in a hang state indefinitely executes instructions without relinquishing the processor. In addition to this fault model, we consider another case of system hang in which the OS halts the processor while waiting for a response from a blocking I/O operation [22] that fails for some reason; or the OS executes a halt instruction, for example due to an error in a control flow of a program.

When the processor is in the halt state, performance monitor counters do not work. However, a timestamp counter still counts clock cycles [22] and can be used to detect system hangs due to halted processor. Upon a context switch, the timestamp counter is set to a fixed timeout value (rather than to an instruction count). The counter decrements at each clock cycle and flags an OS hang when it reaches zero (forcing the system reboot). The fixed timeout value for the hang detection should be set sufficiently large, e.g. 5 seconds, to avoid false alarms, as blocking I/O operations (even when not in error) may take a long time[4].

Check Value Determination. The check value is naturally bounded as the product of the processor speed and the time-slice. A tighter bound is obtained as the running count of the maximum number of instructions executed in a time-slice within a sliding window of a predefined number of time-slices. However, if the system transitions from an idle state to a busy one, the determined check value may be small, causing a false alarm. In this case, a default check value is applied (the product of the processor speed and half the time-slice, in the current prototype. Note that a larger default value reduces false alarms but increases hang-detection latency.

Priority-Based Scheduling. We assume that in a balanced system all tasks have the same priority. Although Linux supports priority-based scheduling, many practical systems usually deal with equal-priority tasks. In these systems, kernel operations (e.g., system calls and interrupt/exception handlers) are completed within a time far less than the process time-slice appropriately selected during the system design. Consequently, SHD is able to detect system hangs successfully. For systems with prioritized tasks, the check values can be set by profiling instruction counts executed in the time-slices of the high-priority tasks.

[4] One may expect to exploit the timestamp counter to detect indefinite execution. But the large timeout value results in long-latency hang detection.

4.2 SHD Implementation

Two pins, P_SCHL (scheduler interceptor) and P_PMC (performance monitor counter), support the SHD implementation depicted in Figure 5. The solid and dotted lines illustrate the control flow of the processor execution. Circled numbers indicate the step order.

An application process, *app1*, enters kernel mode to execute a system call or to handle an interrupt (step 1 in Figure 5). After the kernel mode processing completes, the Linux system checks whether the time-slice assigned to *app1* is used up. If the time-slice is used up, the system invokes the scheduler (step 2). In a normal case, when the RMK is not deployed, the scheduler completes the context switch and transfers the control to the next application process, *app2* (step 9, the dotted line).

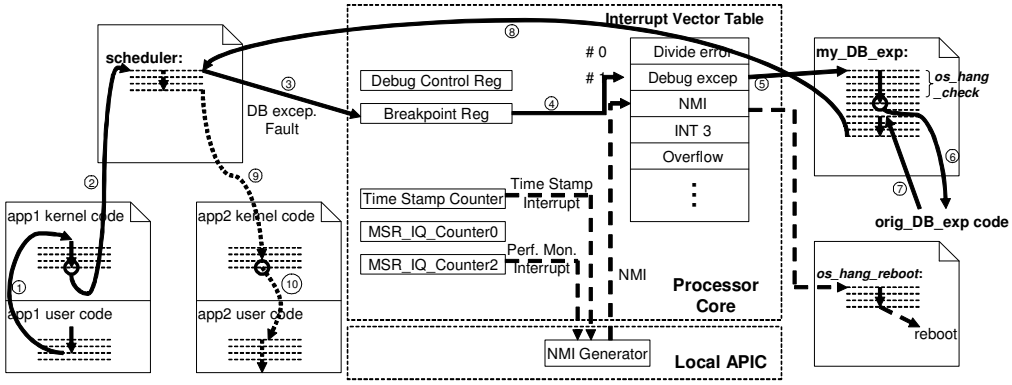


Figure 5: Implementation of the OS hang detection in Linux 2.6.11 on a Pentium 4

When the RMK is deployed, the scheduler entry point (the address of the first instruction of the scheduler) is written into a breakpoint register, and the debug control register is properly configured so that whenever the instruction at the breakpoint is to be executed, a debug exception fault is raised by the processor (steps 3 & 4). The DB exception interrupt handler registered in the system's interrupt vector table points to the custom exception handler, my_DB_exp (step 5). This exception handler performs three tasks: (i) it reads and sets hardware counters in the processor and executes the detection algorithm (i.e. os_hang_check()); (ii) it calls the original debug exception handler (step 6), and (iii) it writes the scheduler entry point into the breakpoint register, configures the debug control register to enable repeating the same processing on the next context switch, and then returns to the scheduler[5] (step 8). The third task is required because the original debug exception handler by default disables the debug exception to be raised. After returning to the scheduler the normal processing resumes (the next process is scheduled) following step 9. Steps 3-8 are provided in the P_SCHL pin with the support of P_DBR and P_INTR.

Two performance counters, MSR_IQ_Counter0 and MSR_IQ_Counter2, are used for profiling and checking, respectively. The Time Stamp Counter is used to handle processor-halt cases. When an OS hang occurs, MSR_IQ_Counter2 decrements to zero and triggers a performance monitor interrupt (PMI) into the NMI generator in the local APIC. The NMI generator raises the NMI to the processor, where the interrupt is converted to the EVT_NMI event and the registered callback function, os_hang_reboot(), reboots the system. The corresponding control flow is depicted by dashed lines in Figure 5.

[5] A *do-not-retrigger* bit in a hardware register should be set here to successfully return to the scheduler. Otherwise, the *debug exception fault* is triggered again upon the return, resulting in infinite recursive fault triggerings.

5. Application Hang Detection Module (AHD)

Applications (or individual application threads) may hang due to hardware faults, deadlock, failed I/Os, etc. The failed thread is either continuously scheduled on the processor (as in an infinite loop) or not scheduled at all (e.g., waiting for wakeup on completion of asynchronous I/O operations). The Application Hang Detection module (AHD) transparently detects application hangs. The AHD exploits the fact that the count of instructions executed in a well-defined code block or code section can be statistically bounded [18][6]. If the instruction count goes outside the bound, 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. After a thread enters a code section, it must leave the section before entering the section again. Examples of code sections include a loop, a loop body (i.e., a single iteration of a loop), a function, and a mutex block (a code block between mutex acquisition and release). A recursively called function is not monitored as a code section. A code section may include multiple nested sections, such s nested loops, and code sections may overlap, 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 simultaneously.

The AHD keeps a logical counter for every thread running in a monitored code section. Consequently, an array of counters monitor a multithreaded application, as shown in Figure 6. When a thread enters a code section, the corresponding counter starts counting instructions from zero, and the counter is called *active*; when it leaves the section, the counter stops counting, becoming *inactive*. Because the thread may be executing an instruction within multiple code sections, multiple counters for the thread may be active at the same time. The granularity of code sections can be selected to limit the number of sections and thus 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, only one hardware performance monitor counter is used for counting. By using the hardware counter and runtime system monitoring, the AHD detects application hangs with low overhead and low latency.

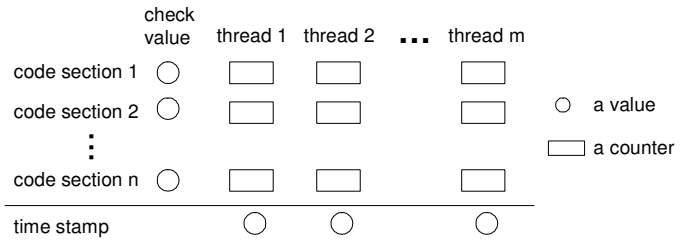


Figure 6: The counter array monitoring a multithreaded application in the AHD

5.1 Issues

Detecting an Unscheduled Thread. A timestamp is introduced for each thread in order to record the time at which a thread has entered a given code section (*active code section*). The AHD performs a periodic check of these timestamps at a fixed interval. If a timestamp expires, the corresponding thread is flagged as hung. Similar to OS hang detection, the fixed interval must be sufficiently large to avoid false alarms, e.g., 5 seconds[7]. Note

[6] Note that the problem of determining the worst case execution time or instruction count of a program or code block is, in general, not decidable and is equivalent to the halting problem.
[7] A thread scheduling may be delayed due to a heavy system load.

that a normal thread may not be scheduled when it executes a function with indefinite waiting time, such as accept(), select(), or recv() in TCP. To avoid false alarms, these functions should not be included in a monitored code section.

Determining Check Value. The check value for a code section is derived from the execution history of the section, similarly to the check-value determination in SHD. However, a code section may have multiple exits, and the counts of instructions for differently exiting executions may vary widely. An example is a switch-case block with a different computation task and a different exit for each case. AHD maintains a history of instruction counts for executions associated with each exit, and the check value for the code section is derived based on the history with the largest instruction counts. A threshold based on heuristics obtained from application profiling is applied to avoid selecting small check values derived from trivial cases of code section execution (e.g., a web server sends a response to a client without sending web pages, because the client has an unexpired copy of the pages.)

5.2 AHD Implementation

The AHD module employs services provided by four RMK pins: P_SCHL, P_SYSC (system call), P_PERI (periodical task) and P_PMC, in a way similar to the SHD implementation. AHD exploits compiler-provided information on code sections to detect application hangs. When generating binary code for a program, the compiler adds two function calls, section_enter(section id) and section_leave(section id, exit id), at the entries and exits of the monitored code sections, respectively. (The compiler may select only coarse-granularity code sections for monitoring, like the outer loops or functions.) A new system call, sys_section_action(), is added to service these two functions.

Only one hardware counter is used in implementing the counter array in Figure 6. For each monitored process, the AHD maintains three kinds of tables: a section table recording all the monitored code sections, a thread-execution table for each thread column in Figure 6, and an instruction-count history table associated with each exit of a code section. The AHD algorithm is outlined as follows:

- *Upon a context switch:*
  - *Read the hardware counter for the number of instructions executed during the last time-slice, and reset the counter to zero.*
  - *The read number is added to the current count values for all the active sections in the thread. If the sum for a section is larger than the section's check value, a hang is detected.*
  - *The timestamp for the thread is set to the current system time.*
- *Upon an instruction retirement:*
  - *The hardware counter increments by 1.*
- *On section_enter(section id) invocation:*
  - *Sets the active bit for the section and creates related table entries if they are not present.*
  - *Reads the hardware counter's value c and sets the current count value of the section within the thread to -c (so that the adding operation performed in the next context switch generates the correct value).*
- *On section_leave(section id, exit id) invocation:*
  - *Reads the hardware counter's value and adds it to the current count value of the section within the thread.*
  - *Appends the sum as a piece of count history in the path table for the section and the exit. Then computes the new check value*[8] *according to the count history for the exit with the largest average instruction count.*
  - *Clears the active bit, and removes unused table entries*
- *Periodically performs the following check at a fixed interval:*
  - *Checks the timestamps for all the threads of the process associating with at least one active section. If a timestamp expired, a hang is detected.*

[8] As checking is not performed here, the sum may be larger than the current check value with the hang notification not raised.

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 that are pending in the system and not finished by 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 schemes, e.g. Plank's libckpt [13], rely 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 requires the program developer to have 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, in 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 OS-level knowledge (e.g., messages can be logged by invoking appropriate operations of the P_NET pin).

In addition to the generic approach to deciding when and how to checkpoint, checkpointing mechanisms can be customized to achieve even higher performance if the application execution characteristics are known. This section discusses how TAC addresses the consistency issue in a generic checkpointing method and how the same approach can be employed to provide a custom application checkpointing. The TAC module is implemented within the Linux RMK framework, and the performance overhead is measured by conducting experiments with real-world application. The Apache web server is used as a target application in our study.

6.1 Generic Checkpointing

Figure 7 illustrates a breakdown of the address space of a process execution in 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 7) 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.

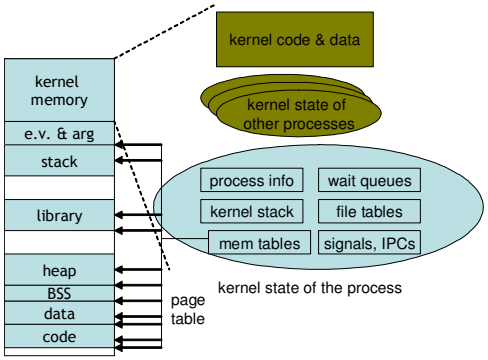


Figure 7: The address space of a process in Linux

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 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).*

Comparison with Shadow-Process Checkpointing. A shadow process [28] may be forked for checkpointing purposes. The shadow process is inactive and only stores the process image. Memory pages are shared between the two processes with the copy-on-write mechanism applied. When the original process fails, the shadow process is started. The TAC has two advantages over the shadow-process method. (i) TAC preserves process state such as process ID and process parenthood. Preserving process ID is crucial for PID-aware applications like the Apache web server. (ii) The shadow-process checkpoint kills the old shadow process and forks a new one during each checkpoint. This incurs a fairly large overhead, including allocation and deallocation of process resources and memory (several milliseconds).

6.2 Custom Checkpointing for Apache Web Server

When the characteristics of an application are used, the generic checkpointing scheme can be customized to a specific application to achieve better performance. We demonstrate this by deploying a custom checkpointing scheme for the Apache web server, a request-transaction-based application.

Apache consists of a parent server process and multiple child server processes, as depicted in Figure 8(a) [25]. The parent server manages the pool of child servers through their process IDs, and the child server performs the web request processing. After a connection from a client is accepted, the child server receives web requests from the client, processes the requests, and sends back replies. If a connection expires when there is no activity during

a pre-specified period, the server waits for the next connection. Figure 8(b) depicts the execution flow as a state transition diagram[9].

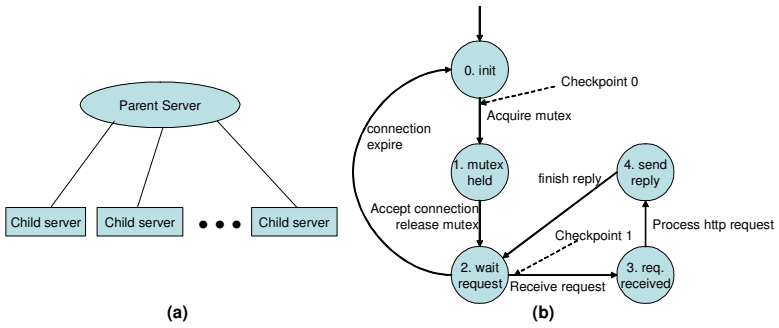


Figure 8: (a) Process architecture of Apache (b) State transition diagram of the child server

When a child server crashes, the connection and the ongoing data transfer are broken, and the web page failure is indicated to a user. If the user initiates the request again, the data must be retransmitted. This incurs a waste of network bandwidth.

The Apache child server state does not depend on the processing history. This feature leads to the custom application checkpointing scheme, which takes only two checkpoints of Apache, *once for each checkpoint throughout the entire life of the application*, and logs the web request and the amount of transmitted data of the incomplete reply. During a failure recovery, one of the two checkpoints is restored with the connection preserved (which checkpoint to restore depends on what state the application is in when the failure occurs), and the logged web request is replayed. Process ID is preserved during the recovery. Since each checkpoint is taken only once throughout the application's life, the runtime overhead of the scheme is negligible.

The details of the transparent custom checkpointing scheme are illustrated in Figure 8(b). Checkpoint 0 is taken the first time the mutex is acquired in state 0; checkpoint 1 is taken the first time a request is received in state 2. When the application fails in state 0 or to state 1, checkpoint 0 is restored and the mutex is released if it is still held. When the application fails in state 2, 3, or 4, checkpoint 1 is restored and the logged request, if there is one, is replayed. During a reply, the part of the reply already sent to the client is not retransmitted.

Implementation. Four pins are employed in the custom checkpointing mechanism for Apache: P_MEM (memory image dumping/restoration), P_FILE (recording the information of opened files), P_NET (intercepting network operations for taking checkpoint 1, logging request, replaying request, and preventing transmission of data already sent), and P_SYSC (system calls for mutex manipulation, including the hook for taking checkpoint 0). The checkpointing and recovery are performed without any instrumentation to the application by employing the operations/events provided by these pins.

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 are conducted on the RMK prototype to evaluate its effectiveness and performance overhead.

[9] Here we only consider processing basic web requests (e.g. static webpage retrievals). More complicated web applications, e.g. e-commerce, may invoke *application servers* and databases, which are beyond this paper's focus.

NFTAPE [24], a software toolset, is used to launch fault injections and assess the effectiveness of individual RMK modules in terms of their error coverage.

7.1 Evaluation of System Hang Detection

Experiment Setup. To assess effectiveness of the OS hang detection we need to create operational conditions that are likely to lead to OS hangs. Toward 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 into the CRC-driver to induce OS hangs. A synthetic application invokes the driver to compute CRC on a selected data set and to generate sufficient system load to enable activation of errors injected into the CRC-driver. The results from the experiments are collected for offline analysis.

Outcome Categories. Outcomes from error injection experiments are classified according to the categories given in Table 4.

Results. Table 5 gives the distribution of fault injection outcomes. Strictly speaking, as errors propagate within the system, a single fault may trigger multiple failures; the first observed failure outcome is reported in Table 5.

Table 5 indicates that 9.0% of injected errors lead to OS hangs and that the OS hang detection provides 100% coverage for the hangs. Table 6 gives five examples of error propagations leading to OS hangs. We 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 OS may be corrupted by invalid instruction execution (GDT, *Global Descriptor Table*, and TSS, *Task Stack Segment*, in the system are corrupted in examples 4 and 5). Surprisingly, kernel exceptions may be triggered repeatedly (e.g., spin_lock_unreleased is reported more than 30 times before GDT is corrupted in example 5).

Table 4: 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 5: 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.

**This number includes the case marked with *.

Table 6: Examples of error propagations leading to OS hangs

Example	First Observed Failure 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

SHD detects a hang after a predetermined number of instructions executed without a context switch, then reboots the system. Therefore, it is not known how long after the hang detection the system remains in a non-operational 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 a 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 a kernel exception or not trigger an exception but continue hanging. This study also indicates that, although the OS 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 complementary solution for detecting erroneous instruction executions with a bounded latency.

7.2 Evaluation of Application Hang Detection

Experiment Setup. The web server Apache 2.0.55 is the target application for fault injection to evaluate the AHD. A website consisting of hundreds of pages and files is set up on the target machine. A web client launches multiple requests from another machine. Since the application consists of tens of thousands of lines of code, fault injection is conducted into the main loop of the server, which accepts web requests and sends back replies (illustrated in Figure 8(b)). Three code sections are identified in the target code: *initialization* (from state 0 to state 1 in Figure 8(b)), *connection acceptance* (from state 1 to state 2), and *request processing* (states 2, 3 & 4). The instruction counts of 50 recent executions are used to determine the check value for each code section.

Results. Three fault injection campaigns, listed in Table 7, are launched to evaluate the AHD. In each experiment of the three campaigns, the AHD is trained with 50 web requests to learn the initial check value. The threshold for determining the check value in all the three campaigns is selected as 1.5 million instructions (equivalent to a 1ms-duration execution of the processor) to avoid small check values derived from trivial cases of code-section execution.

Table 7: Experiment campaigns for application hang detection

Campaign	Injected Fault	Experiments	Crashes	Detected Hangs
1	A bit flip in the focused code	1861	1854 (99.6%)	7 (0.4%)
2	A hang in the focused code	1206	0	1206 (100%)
3	None	2000	0	1

Bit flips are injected into the target code in Campaign 1. Table 7 shows that the application is very likely to crash (99.6%) in the presence of bit errors. Only seven of the injections lead to hangs. As seven hangs are not sufficient to evaluate the detection coverage of the AHD, hangs are explicitly injected into the target code in the

second campaign. A hang is injected by activating an infinite loop instrumented in the application for fault injection purposes only. The coverage of the application hang detection is 100%.

Campaign 3 was launched to measure false positives, and hence, no faults are injected in this campaign. In the 2000 experiments, only one raises a false positive. The false positive was raised because a very large file was requested following a number of requests for small pages/files. The transmission of this large file requires that many more instructions be executed within a code section (2726433 instructions are executed for the request processing, while the largest instruction count for the previous 50 request processings is 449493). The false positive may be avoided with a longer monitoring history (e.g. 100 requests), or a higher threshold[10].

7.3 Performance Overhead of RMK, SHD, and AHD

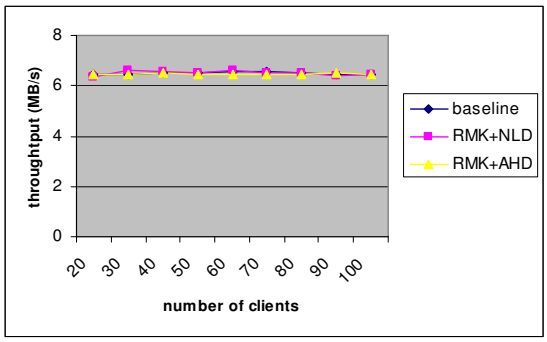
Experiment Setup. Experiments are conducted to study the performance overhead of the RMK as well as individual modules. Two Linux machines are used in the experiments: one as the web server and the other as the web client. The Apache 2.0.55 runs on the server machine. The web clients on the client machine are launched by WebStone [20], a well-known benchmark for web servers. WebStone creates a load on a web server by simulating the activity of multiple web clients on one or more machines. In our experiments, WebStone starts 20-100 simulated web clients on the client machine. Three suites of experiments are carried out: baseline (RMK not deployed), RMK+SHD (RMK with the SHD module deployed on the server machine), and RMK+AHD (RMK with the AHD module deployed to detect hangs of the Apache server). No fault injection is performed in the experiments.

Results. Figure 9 gives the server throughput (amount of data transmitted per second) and average response time for a request as a function of number of clients. One can see that the throughput does not change with the number of clients, and the average response time increases linearly with the number of clients. This is because, the web clients launched by WebStone send requests to the server continuously, and the server processes the requests at its full throughput, which is a fixed value. Because the server's processing capability is fixed, the request response time gets larger when more clients send requests.

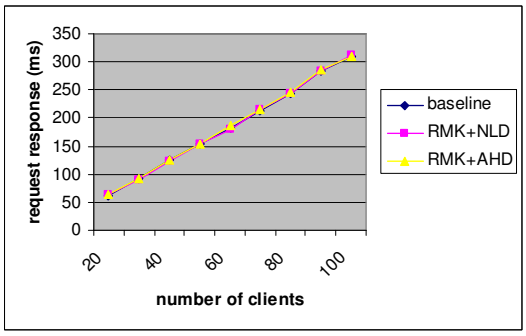
Figure 9 shows that the performance overheads of RMK, SHD, and AHD are very small. The actual throughput (with the 95% confidence interval) measured for the three scenarios in Figure 9(a) is 6.51±0.04ms for the baseline, 6.51±0.07 for RMK+SHD, and 6.47±0.02 for RMK+AHD.

The overhead of SHD is negligible because instruction counting is performed in hardware and the context switch frequency is not high. AHD incurs an overhead of about 0.6% of the baseline performance. The AHD overhead is due to (i) system call invocations at the code section entry/exit points, (ii) updates to related count values at context switches, and (iii) periodical timestamp checks. Concerning the low frequencies of context switches and timestamp checks, (ii) and (iii) incur negligible overheads. If code sections are entered and exited frequently, the overhead in (i) is noticeable, but still small.

[10] Prior knowledge of the size of the requested file/page can be exploited to reduce false positives.



(a) Throughputs of the web server



(b) Average response time for a request

Figure 9: Performance overheads of RMK and SHD/AHD modules

7.4 Performance Overhead of TAC

Three applications are used to evaluate the performance overhead of the generic checkpointing in TAC: (i) gzip, a SPEC2000 benchmark, (ii) ns2, a popular network simulator, and (iii) Apache, a web server. Because the TAC module aims at providing high-efficiency checkpointing for computation-intensive applications, gzip and ns2 are selected as target applications in the experiments. Apache is included to contrast the generic checkpointing scheme with a custom checkpointing approach. Experiment results show that TAC incremental checkpointing has a 0.1% performance overhead and gains a 91% performance improvement against the memory-dump approach (dumping all process pages to backup memory) for a large-memory application. These results are consistent with the measurements in libckpt [13] (85% of overhead reduction) and libFT [31] (0.1% performance overhead).

7.4.1 Experiment Results for Generic Checkpointing

As an in-memory checkpointing scheme, the TAC has a relatively small performance overhead, and more frequent checkpointing can be applied to avoid a large loss of work due to application failures (the checkpoint interval for the experiments is 2 seconds). Table 8 summarizes the TAC performance overheads (the 95% confidence interval) and compares the results with the performance of the memory-dump approach.

Table 8 lists the numbers of data pages and pages written during a checkpoint interval for the three applications. The breakdown information (in parenthesis) of pages accessed during a checkpoint interval provides insights into the executions of the applications. For example, gzip accesses a total of 111 user-space data pages, including initialized static data (data), uninitialized static data (bss), dynamic data (heap), stack, and data of shared libraries. Apache uses 321 pages, and ns2 uses 1426 pages, the largest number among the three benchmarks. This is because gzip uses a streaming mode in data compression (read a block – compress – write result – read next block – …), and a small amount of fixed-size memory is sufficient (gzip has no heap). In the conducted experiments, ns2 simulates a network transmission protocol in a complex network configuration and explores a large state space in the simulation, so the memory requirement is high.

Due to space locality, the number of memory pages written by an application within a short period is often far less than the total page number. In our experiments, around 30 pages are written in the three applications during a checkpoint interval, though the total page numbers ranges from several hundred to several thousand.

TAC checkpointing involves two activities: (i) periodically setting the entire process memory to read-only (the associated overhead is denoted as *c* in Table 8) and (ii) upon a page fault, raising the page fault exception (the overhead denoted as *s*), granting write access to the targeted page, and replicating the page (denoted as *r*). The

TAC performance overhead is computed by adding the overheads of the two activities. Let p denote the number of page faults in a checkpoint interval. Then the TAC checkpoint overhead during a checkpoint interval is:

$$TAC\_ovhd = c + (s+r)*p \ .$$

Table 8: TAC performance overhead during a checkpoint interval (2 seconds)

Application	Data Pages, *n*	Written Pages, *p* *	TAC Checkpoint Overhead (ms)			Memory-Dump Overhead (ms)
			Setting pages as read-only, *c*	Handling a page fault, *s+r*	Overall, *TAC_ovhd*	
gzip	111 (data: 2, bss: 83, heap: 0, e.v.+arg+stack: 21, lib data: 5)	29±6 (data: 1, bss: 27, heap: 0, stack: 1, lib data: 0)	0.0083±0.0043	0.0344±0.0025	1.01	1.94±0.16
Apache	321 (data: 7, bss: 3, heap: 203, e.v.+arg+stack: 21, lib data: 87)	32±8 (data: 5, bss: 1, heap: 17, stack: 3, lib data: 6)	0.0191±0.0036	0.0344±0.0025	1.12	3.10±0.15
ns2	1426 (data: 158, bss: 8, heap: 1213, e.v.+arg+stack: 21, lib data: 26)	34±6 (data: 1, bss: 1, heap: 31, stack: 1, lib data: 0)	0.1182±0.0034	0.0344±0.0025	1.29	15.08±0.10

*The breakdown information of written pages listed in the column is collected from a sample page-access trail in a checkpoint interval.

The measurement in the experiments shows $s = 24.55 \pm 2.02\mu s$, and $r = 9.86 \pm 0.46\mu s$ (with the 95% confidence level). Because s measures triggering a page fault as well as delivering a signal, and r measures copying one page and granting write permission for the page, s and r have the same values for the three benchmark applications. The values of c are different for the three applications because different numbers of memory pages (denoted as n) are set as read-only. The TAC_ovhd values listed in Table 8 are computed using the corresponding average values of *c, s, r,* and *p.*

Comparison with Shadow-Process Checkpointing. Both TAC checkpointing and shadow-process checkpointing have overheads computed by $c+(s+r)*p$, but the c in shadow-process checkpointing is much larger because a process forking and a process termination are included in the checkpointing (for the Apache application, the overhead associated with the process forking and termination is 1.75 ± 0.05ms, much larger than the 0.019ms in TAC).

Comparison with Memory-Dump Checkpointing. The overall overheads for the three benchmark applications, 1.01ms, 1.12ms and 1.29ms, are less than 0.1% of the checkpoint interval (2 seconds) and justify the selection of checkpoint intervals of several seconds. Table 8 (the rightmost column) provides comparison of the TAC checkpoint overhead with the overhead of the memory-dump method. One can see from the table that TAC checkpointing has a great performance improvement than memory-dump checkpointing, especially for applications with large memory (1.29ms versus 15.08ms, a 91% improvement for ns2). This is because TAC effectively exploits the space locality of memory accesses performed by applications.

7.4.2 Experiment Results for Custom Checkpointing for Apache

Experiments are conducted to evaluate the custom checkpointing for Apache provided in TAC. The performances of three different checkpoint/recovery approaches are compared in the experiments: i) a simple restart solution, ii) memory-dump checkpointing, and iii) custom checkpointing for Apache. The comparison in terms of the runtime checkpoint overhead and recovery time is summarized in Table 9 (with a 95% confidence level).

From Table 9, one can see that if there is no checkpointing provided for Apache, the recovery time is large (42.02ms), which greatly degrades web service availability. The recovery times for memory-dump checkpointing and custom Apache checkpointing are similar (around 4ms), as both recover the application from in-memory checkpoints. But the runtime overhead of the custom Apache checkpointing is very small (not measurable in the system).

Table 9: Comparison of three checkpoint/recovery approaches

Approach	Description	Checkpoint/Logging Overhead (ms)	Recovery Time (ms)
Simple Restart	The web server is restarted upon failure. No checkpointing	N/A	42.02 ± 0.13
Memory-Dump	The process image is dumped to backup memory.	3.10 ± 0.15	4.05 ± 0.08
Custom Checkpointing for Apache in TAC	Two checkpoints are taken only once during execution. Information logging is performed at runtime.	Not measurable*	4.07 ± 0.07

* The time-measuring tool provided by the system is not able to measure so small an overhead, and it reports 0.

8. Conclusions

The paper describes the Reliability MicroKernel (RMK) framework, a loadable kernel module for providing application-aware reliability, and the configuring reliability mechanisms installed in the 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 shows that the OS hang detection and application hang detection, which exploit characteristics of application and system behavior, can achieve 100% coverage and low false-positive rates (1 out of 2000 experiments for application hang detection, and no false positive for OS hang detection). Because the OS-level knowledge of applications and of the system is used, the RMK prototype has a low overhead in providing transparent application checkpoint (less than 0.1% in the experiments) and application failure detections (0.6% performance overhead). In the future, we will focus on the design and deployment of more advanced reliability and security mechanisms, including monitoring the memory locations accessed by individual application functions for detecting memory corruption, enforcing application launching regulation for detecting malicious attacks (e.g., a web server never launches a shell), and tracing system calls invoked within specific application functions for detecting system anomalies.

References

[1] D. Bovet et al., *Understanding the Linux Kernel*, 3rd Edition, Chapter 6.1, "Clock and Timer Circuits," O'Reilly & Associates, Inc., 2005.

[2] Trace Distribution Center, Performance Evaluation Laboratory, Brigham Young University, http://tds.cs.byu.edu/tds.

[3] "Open, Secure, Scalable, Reliable UNIX for POWER5 Servers," AIX 5L for POWER Version 5.3, IBM Corporation 2004.

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

[5] Mark Russinovich and Zary Segall, "Fault-Tolerance for Off-The-Shelf Applications and Hardware," Proc. of the 25th International Symposium on Fault-Tolerant Computing, 1995.

[6] R. Rashid et al., "Mach: A Foundation for Open Systems," in Proc. 2nd Workshop Workstation Operating System. Sept. 1989.

[7] M. Rozier et al., "Overview of the CHORUS Distributed Operating Systems," USENIX Workshop on Microkernels and Other Kernel-Architectures, Seattle, WA, 1992.

[8] Nithin Nakka et al., "An Architectural Framework for Detecting Process Hangs/Crashes," EDCC 2005: 103-121.

[9] Nithin Nakka et al., "An Architectural Framework for Providing Reliability and Security Support," DSN 2004: 585-594.

[10] *Operations Manual: PCI Watch-Dog Timer Card*, the company of DECISION-COMPUTER Deutschland, http://www.decision-computer.de/dowSHD/sonstiges/manualPCIwatchdog.PDF.

[11] Daniel Beauregard, "Error Injection-Based Failure Profile of the IEEE 1394 Bus," MS thesis, 2003.

[12] Zbigniew Kalbarczyk et al., “Application Fault Tolerance with Armor Middleware,” IEEE Internet Computing 9(2): 28-37, 2005.

[13] J. Plank et al., “Libckpt: Transparent Checkpointing under Unix,” Conference Proceedings, Usenix Winter 1995 Technical Conference, New Orleans, LA, January, 1995, pp. 213—223.

[14] Steven Osman et al., “The Design and Implementation of Zap: A System for Migrating Computing Environments,” ACM SIGOPS Operating Systems Review, Volume 36, 2002.

[15] E. Pinheiro, "Truly Transparent Checkpointing of Parallel Applications,” Internal Report, Dec. 1997. http://www.research.rutgers.edu/~edpin/epckpt/epckpt.ps.gz.

[16] Long Wang et al., “Checkpointing of Control Structures in Main Memory Database Systems,” DSN 2004: 687-692.

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

[18] 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).

[19] M. Russinovich et al., *Microsoft Windows Internals*, 4th Ed., Microsoft Press, 2005.

[20] G. Trent and M. Sake, *WebSTONE: The First Generation in HTTP Server Benchmarking*, MTS Silicon Graphics, Feb. 1995.

[21] A. Chou et al., “An Empirical Study of Operating Systems Errors,” Symposium on Operating Systems Principles, 2001.

[22] *IA-32 Intel® Architecture Software Developer’s Manual, Volume 3: System Programming Guide*, Intel Corp., 2006.

[23] *IA-32 Intel® Architecture Software Developer’s Manual, Volume 1: Basic Architecture*, Intel Corp., 2006.

[24] 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.

[25] L. Stein et al., *Writing Apache Modules with Perl and C*, O'Reilly & Associates, Inc., 1999.

[26] F. Salles et al., “MetaKernels and Fault Containment Wrappers,” Proc. FTCS-29, pp. 22-29, 1999.

[27] J. Arlat et al., “Dependability of COTS Microkernel-Based Systems,” IEEE Transactions on Computers, Vol. 51, No. 2, Feb. 2002.

[28] S. Srinivasan et al., “Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging,” USENIX Annual Technical Conference, General Track, pages 29--44, 2004.

[29] P. Bernstein, “Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing,” IEEE Computer, Feb. 1988.

[30] W. Gu et al., “Characterization of Linux Kernel Behavior under Errors,” DSN 2003.

[31] Y. Huang et al., “Software-Implemented Fault Tolerance: Technologies and Experience,” Proc. IEEE Fault-Tolerant Computing Symp., June 1993.

[32] N. Neves et al, “RENEW: A Tool for Fast and Efficient Implementation of Checkpoint Protocols,” Proc. of the 28th International Symposium on Fault-Tolerant Computing (FTCS), June 1998.

[33] L. Wang et al, “Modeling Coordinated Checkpointing for Large-Scale Supercomputers,” DSN 2005.

[34] M. Beck et al., “Compiler-Assisted Checkpointing,” Technical Report UT-CS-94-269, 1994.