Reliability MicroKernel: Providing Application-Aware Reliability in the OS

Long Wang, *Student Member, IEEE*, Zbigniew Kalbarczyk, *Member, IEEE*, Weining Gu, *Member, IEEE*, and Ravishankar K. Iyer, *Fellow, IEEE*

***Abstract*—This paper describes the Reliability MicroKernel (RMK) framework, a loadable kernel module (or a device driver) for providing application-aware reliability, and dynamically configuring reliability mechanisms. Characteristics of application/system execution are exploited transparently through application-aware reliability techniques to achieve low-latency detection, and low-overhead checkpointing. The RMK prototype is implemented in both Linux, and Windows; and it supports detection of application/OS failures, and transparent application checkpointing. Experiment results show that the system hang detection and application hang detection, which exploit characteristics of application, and system behavior, can achieve high coverage (100% observed in our experiments) with a low false positive rate. Moreover, the performance overhead of RMK, and its detection/checkpointing mechanisms, is small: 0.6% for application hang detection, and 0.1% for transparent application checkpointing in the experiments.**

***Index Terms*—Application aware reliability, OS-level error detection, system crash/hang detection, transparent application checkpointing.**

ACRONYMS[1]

RMK	Reliability MicroKernel
SHD	System Hang Detection
AHD	Application Hang Detection
TAC	Transparent Application Checkpointing

I. INTRODUCTION

OPERATING systems enable the collection, and extraction of rich information on application execution characteristics, including counts of executed instructions, program counter traces, memory access patterns, and OS-generated signals. The 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], their 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 device driver, 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. The implemented RMK modules include OS/application hang/crash detection, and application transparent checkpoint. The architecture exploits processor-level features (debugging, and monitoring facilities available in the current generations 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 the 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 on the specifics of the OS, or the underlying hardware. Several techniques have been implemented to demonstrate how OS knowledge on application execution can be used to provide error detection, and recovery. Table I briefly summarizes the techniques currently available in the RMK.
- **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.
- **Platform independence of the RMK architecture, and modules**. The architecture of the RMK is independent of which platform (hardware, and operating system) the RMK is on. Moreover, the set of RMK modules implemented on an operating system (e.g. Linux) can be deployed onto other systems (e.g. Solaris, FreeBSD, or Windows systems) with minimal, or no code changes, as long as corresponding RMK pins (the platform-dependent components)

Manuscript received January 15, 2007; revised May 1, 2007; accepted June 3, 2007. This work was supported in part by NSF grants CNS-0406351 (Next-generation Software), CNS-05-24695, CNS-05-51665, and ACI-0121658 ITR/AP, the Gigascale Systems Research Center (GSRC/MARCO), and Motorola Corporation as part of Motorola Center. Associate Editor: Y. Dai.

The authors are with the Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, IL 61801 USA (e-mail: longwang@crhc.uiuc.edu; kalbar@crhc.uiuc.edu; wngu@crhc.uiuc.edu; iyer@crhc.uiuc.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2007.909758

[1]The singular and plural of an acronym are always spelled the same.

0018-9529/$25.00 © 2007 IEEE
Authorized licensed use limited to: Tsinghua University. Downloaded on August 14,2026 at 07:47:49 UTC from IEEE Xplore. Restrictions apply.

TABLE I
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. If the application fails while the system is still operational, the original pages are restored.
System 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 system hangs, it does not schedule processes, and the instruction count goes beyond the preset scope.

are implemented, and compiled for the target operating system. Currently, the RMK has been implemented on both Linux, and Windows systems.

Fault/error injection experiments conducted to evaluate the efficiency of the RMK implementation show that the OS-level mechanisms detect all application, and system 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 system hang detection).[2] Additionally, the RMK-based mechanisms provide low-latency detection, and transparent checkpointing with little impact on system performance (0.6% performance overhead for the application hang detection, and 0.1% overhead for the transparent application checkpointing).

II. RELATED WORK

Table II 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, rather than explicitly focus on reliability. Reliability architectures in IBM AIX [3], and High-Availability Linux [4] are designed to be closely coupled with the systems. They mostly concern 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.

Most existing techniques for detecting hangs of the OS, and/or applications, are timer-based; and the timeouts are fixed values that are either preset, or derived from profiling. As a result, the timeout values are often very conservative, and cause large detection latency. Some existing detection mechanisms cannot operate when the interrupts are disabled due to system hangs, e.g., KHM [11]. Others, like the Linux NMI watchdog timer [17], do not detect hangs if the timer interrupt continues to function despite the system hang.

As there are many checkpoint techniques, we do not list all of them in Table II. Some checkpoint mechanisms preserve the entire process image for failure recovery, or process migration [14], [15], while others provide incremental checkpointing [13], [28], [29], [31]. Libckpt [13] checkpoints dirty pages of a process. Mechanisms in [13], and [31] allow users to specify critical data for checkpointing; these 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 states. 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 failures. Our checkpointing scheme solves this inconsistency.

III. RMK FRAMEWORK

The Reliability MicroKernel (RMK) framework is developed as a loadable kernel module (or a device driver) in the operating system. The RMK has a two-level hierarchy, shown in Fig. 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. Well-defined pin interfaces (available to users) can be selectively used to build RMK modules. Each RMK module implements a specific error detection or recovery mechanism, e.g., crash detection, and recovery of applications; hang detection of applications; hang detection of the operating system; or transparent application checkpoint, and recovery. The RMK core between the two levels manages the installation, and de-installation of pins, and modules; it also invokes pin functionalities on behalf of RMK modules. A set of RMK API is provided by the RMK core for the purpose of management.

The abstraction introduced by the two-level RMK structure enables: i) the use of standard operating system functions to create service essentials in designing, and implementing reliability mechanisms (encapsulated as RMK modules); ii) the portability of RMK modules across operating systems, because pins implemented on different platforms export the same interfaces (no need to modify the module code); and iii) transparent coordination between multiple modules, and pins to avoid potential conflicts. For example, two RMK modules, system hang

[2]The 100% coverage, and the false positive rate of 1 out of 2000 are the observed results of the conducted experiments.

Authorized licensed use limited to: Tsinghua University. Downloaded on August 14,2026 at 07:47:49 UTC from IEEE Xplore. Restrictions apply.

TABLE II
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 OS.
	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 OS. 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]	System hang detection: an external PCI card is used as watchdog.	Extra hardware is required; long latency (timeout sometimes in minutes)
	KHM [11]	System hang detection: a process periodically sets a mark, and the timer interrupt handler checks the mark to detect system hangs.	Fails when interrupt is disabled; long latency when system is heavy loaded; large overhead.
	Linux NMI Watchdog [17]	System hang detection: if there is no timer interrupt within a few seconds, the system hang is detected.	Fails when the system 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.

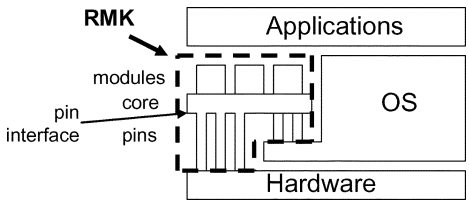


Fig. 1. The RMK in the system.

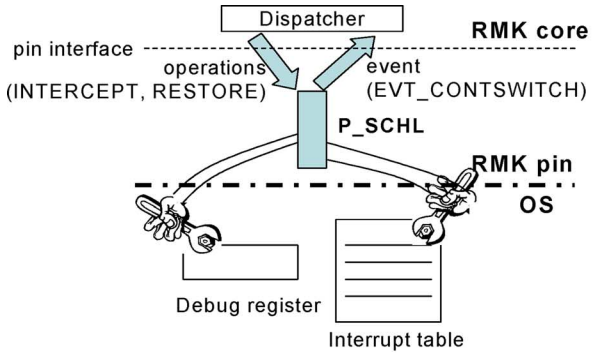
detection, and application hang detection, both intercept process context switches to detect hangs. The RMK coordination mechanism handles scenarios such as the installation of application hang detection while system hang detection is already in place, and allows the two modules to function without conflicts.

A. 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 that is essential in providing reliability mechanisms. In this sense, a pin implementation is platform-specific. But while the pin implementation on different platforms may differ, the pin functionality, and exported interface are intended to be the same, regardless of the platform.[3] For example, Fig. 2 depicts the architecture (Fig. 2(a)), and implementation (pseudocode in Fig. 2(b)) of the *P_SCHL* pin in Linux to intercept the process

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

Authorized licensed use limited to: Tsinghua University. Downloaded on August 14,2026 at 07:47:49 UTC from IEEE Xplore. Restrictions apply.



(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

Fig. 2. The implementation of an example pin, P_SCHL in Linux.

context switch, which is an essential service in enabling application, and/or system hang detection.[4]

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 of intercepting 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 instrumentation of the operating system source code.

Generically, the interface exported by a pin consists of the following two sets.

- A set of operations the pin can perform. In the case of P_SCHL, the operations include INTERCEPT, and RESTORE (see Fig. 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 Fig. 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 Pentium 4), thus forcing the processor to raise a debug exception whenever *schedule()* is invoked[5]; 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 the context switch.

A small set of RMK API *(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 maintained by the dispatcher (see Fig. 2(a)), and populated at the time of module initialization.

[4]In the Windows operating system, the pin interface for P_SCHL is the same, but the implementation is slightly different.

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

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 & 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 in Linux 2.6.11.3). Table III lists the RMK pins currently implemented in the RMK prototype.

B. Application-Level RMK Interface

Applications are monitored by RMK modules, which implement application-specific detection & 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.

C. RMK Core

The RMK core is responsible for: i) maintaining mapping between the events generated by pins and the modules subscribed to these events, ii) delivering the events to the modules, iii) dispatching operation requests to the pins, and iv) managing and configuring the pins, and modules (e.g., installation/removal). The RMK core consists of four components, illustrated in Fig. 3, and listed here.

- ***Pin manager*** is responsible for registration, and loading/unloading of RMK pins.
- ***Module manager*** handles installation, and removal of RMK modules.
- ***Dispatcher*** maintains subscription information for events generated by pins (i.e., mapping between events and RMK

Authorized licensed use limited to: Tsinghua University. Downloaded on August 14,2026 at 07:47:49 UTC from IEEE Xplore. Restrictions apply.

TABLE III
RMK PINS IN THE RMK PROTOTYPE

RMK Pin	Hardware/OS Features	Pin Functionalities
P_PMC	Hardware counters available in modern processors; permit monitoring and measuring processor performance parameters, including instruction count, TLB access, and cache references.	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) provided in modern processors. APIC 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 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	OS-level interrupt handling.	Installs hooks for specific interrupts; generates an event upon an interrupt.
P_SIG	Signals delivered by the operating system to processes.	Intercepts particular signals to processes; generates an event upon a signal delivery.
P_MEM	Memory management provided in the OS.	Copies memory pages; sets page properties.
P_SCHL	Process context switch supported by OS scheduler.	Intercepts context switches; generates an event upon a context switch.
P_PERI	System-level periodic jobs 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 performed by the system.	Intercepts network activities, including message sending and arrival; generates events upon these activities.
P_SYSC	A variety of system calls executed by the system on behalf of applications.	Intercepts specific system calls; generates an event upon a system call.
P_FILE	Functions to manage the table of open files and read/write operations.	Wraps the manipulation of the file table; generates an event when a monitored file read/write completes.
P_IPC	IPC mechanisms, such as pipes, message queues, and semaphore.	Wraps the manipulation of IPC; generates an event when a monitored IPC occurs.

* The P_DBR pin handles use of debug registers for debugging. When the debugged application is switched off the CPU, the pin saves the current value of the registers, and sets new values. When the debugged application is scheduled onto the processor, the pin restores the saved register value. In cases when debug registers are used to intercept process context switch (like the system hang detection module on Linux), a binary rewriting technique can be applied to implement context switch interception, and the debug registers are not used (like the system hang detection module on Windows).

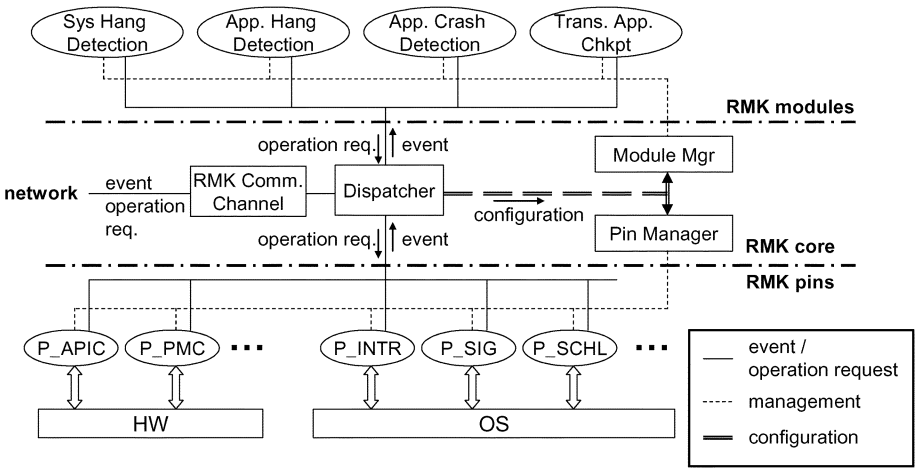


Fig. 3. RMK Architecture.

modules), and delivers the events to modules according to the subscription list.

- ***RMK communication channel*** 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.

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 Fig. 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 system hang detection module,* which loads five pins (P_SCHL, P_PMC, P_APIC, P_DBR, P_INTR) into memory. A new

Authorized licensed use limited to: Tsinghua University. Downloaded on August 14,2026 at 07:47:49 UTC from IEEE Xplore. Restrictions apply.

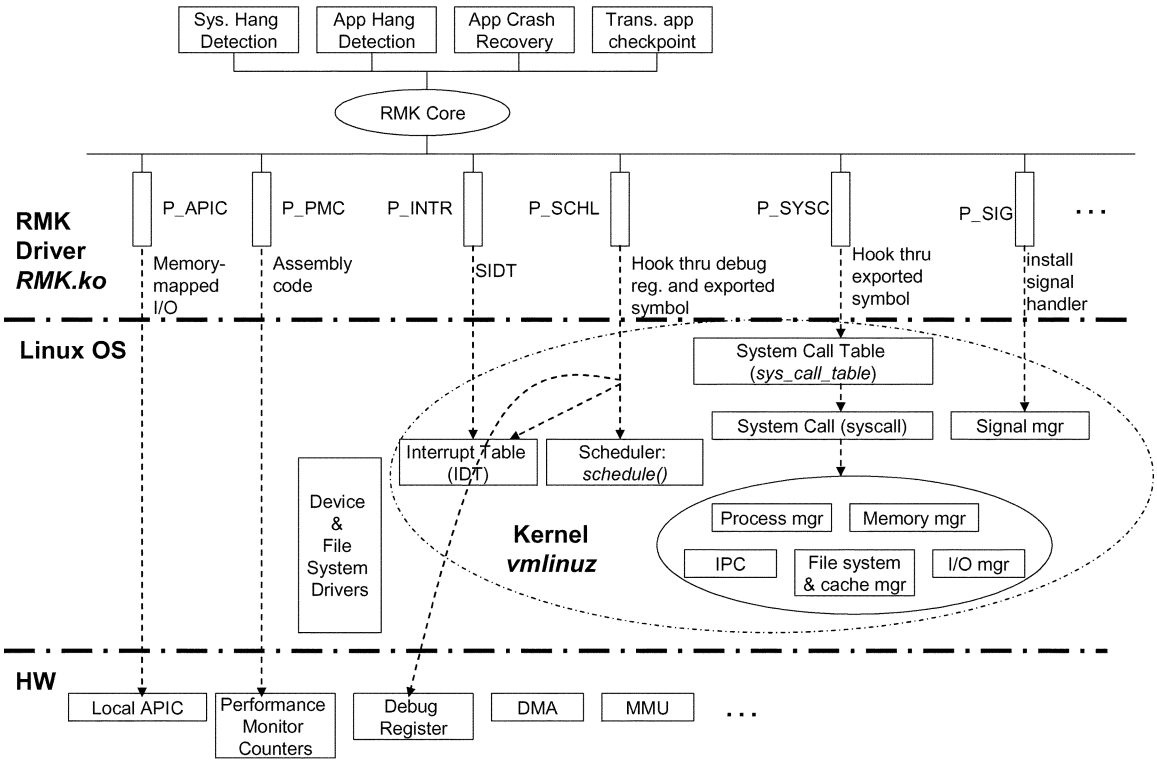


Fig. 4. The RMK Implementation, and Deployment on Linux 2.6.11.

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, and P_SYSC. Now, during initialization of the AHD module, services from these six pins are requested via the RMK API. 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. Recall that the pin manager maintains the state of all the pins in the local host indexed by the pin ID. 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 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).

IV. RMK Implementation on LINUX, and WINDOWS

The decoupling of RMK modules (platform-independent), and RMK pins (platform-dependent) allows for implementing the RMK architecture, and RMK modules only once (except for the RMK modules with platform-specific semantics), and re-implementing the RMK pins accordingly on different platforms. RMK pins are small in terms of the code size (around a few hundred lines for each pin). Figs. 4 and 5 depict the implementation, and deployment of the RMK on Linux 2.6.11, and Windows XP Professional (SP2), respectively. In both figures, the RMK is deployed as a loadable kernel module (or a device driver). Construction of a set of sample RMK pins is also shown in the figures.

The RMK on Linux, and Windows (*RMK.ko,* and *RMK.sys*, respectively) have the same architecture, and RMK modules have the same implementation without major code changes. However, the implementation of RMK can be: i) OS dependent if a pin uses OS specific services, or ii) OS independent if a pin interacts only with the hardware rather than with the OS. The rest of this section discusses examples of pins from the two categories.

For example, the implementation of the P_SCHL pin, which intercepts process context switches, and issues the EVT_CONTSWITCH event, is OS-dependent. The context switch routine on Linux, *schedule()*, is exported by the Linux kernel, and can be intercepted by setting a breakpoint at the entry of *schedule()*. Whenever *schedule()* is invoked, a breakpoint handler is invoked, which generates EVT_CONTSWITCH, and delivers it to the RMK core. On Windows, the context switch routine, *SwapContext()*, is not exported, and can be intercepted by applying the *kernel inline hooking* technique [35], which allows

Authorized licensed use limited to: Tsinghua University. Downloaded on August 14,2026 at 07:47:49 UTC from IEEE Xplore. Restrictions apply.

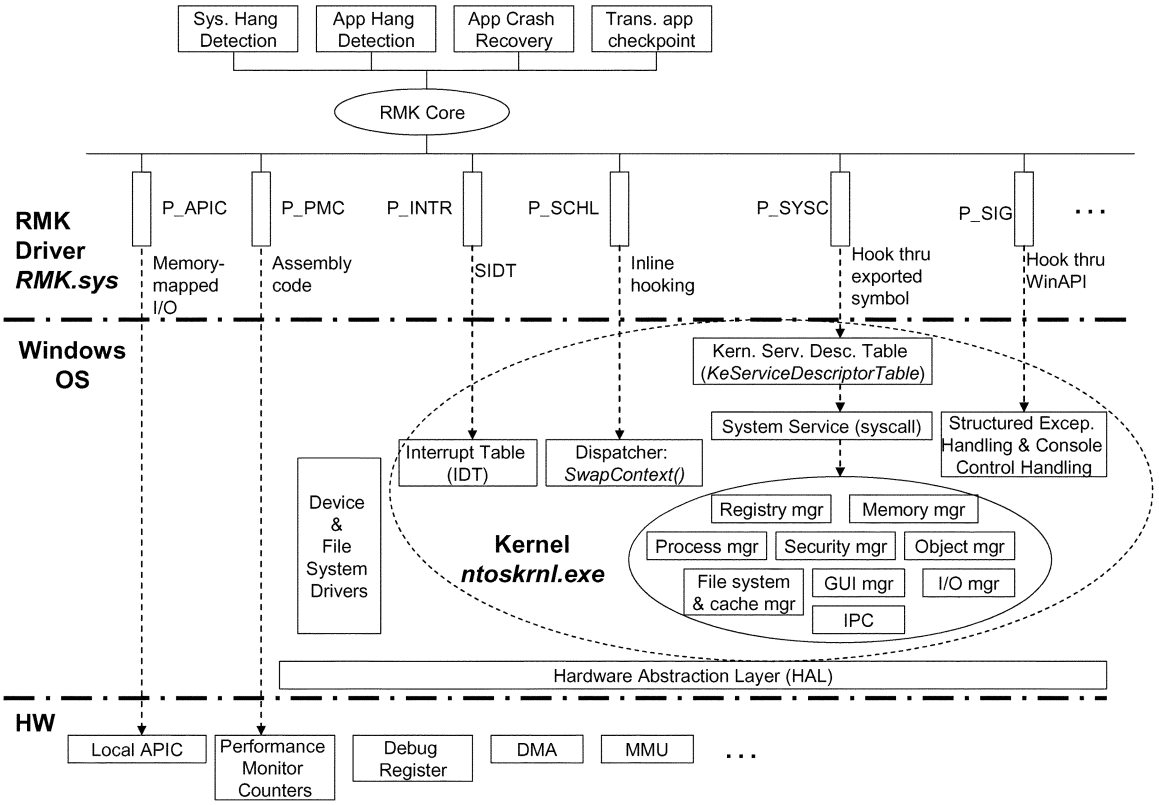


Fig. 5. The RMK Implementation, and Deployment on Windows XP Professional.

searching for predefined instruction patterns in the kernel code space (*ntoskrnl.exe*). The technique is used to search for a sequence of instructions found at the beginning of the *SwapContext()*. These instructions are the same for all Windows XP systems, and can be obtained by analyzing the *ntoskrnl.exe* file offline. After locating the *SwapContext()*, the first instruction of *SwapContext()* is replaced with an unconditional jump to the hook function[6]; and at the end of the hook function, the replaced instruction is executed, and the control flow jumps back to the instruction in *SwapContext()* that follows the replaced instruction.

Another pin with different implementations on Linux versus Windows is *P_SYSC*, which intercepts system calls. Although the system call tables on both of the platforms (*sys_call_table* in Linux, and *KeServiceDescriptorTable* in Windows) are exported by the kernel, they each have specific data structures which require different handlings.

The P_APIC, and P_PMC pins can be reused without a reimplementation because they interact with the hardware instead of the operating systems. Similarly, the P_SCHL pin for Windows can be reused in any Windows system on different hardware. Also note that RMK pins do not need to be reimplemented across variants of a platform. So the implementation of the P_SCHL pin on Linux 2.6 works on Linux 2.4, and the implementation of the P_SCHL pin on Windows XP works on the Windows 2003 Server as well.

[6]If the size of the first instruction is less than the size of the jump instruction, two or more instructions are replaced.

RMK use scenario. RMK-supported services are portable across different platforms (hardware, and operating systems). As a result, a broad range of applications can potentially benefit from RMK. Consider a cluster of web servers on a number of heterogeneous nodes (Linux, and Windows). These web servers may crash or hang due to many causes including corrupted (or invalid) input, or bugs in poorly written device drivers. With the RMK installed on every node, the server downtime, and the system downtime can be greatly reduced; and the availability of web services can be largely enhanced. With the platform-independence of RMK modules, the RMK can scale to embedded devices, e.g., smart phones, and enhance reliability without major change to module design and implementation. Furthermore, RMK-based services can be extended to provide the protection against malicious tampering with the system, e.g. monitoring access to critical application data, or system resources.

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

V. System Hang Detection Module (SHD)

The operating system is subject to hangs due to, e.g., poorly written drivers with blocking operations,[7] and unreleased locks. 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

[7]For example, a device driver, after interrupt is disabled, performs a blocking operation to acquire a lock that is held by another process. As the blocking mutex-acquisition operation is performed in the kernel, the system hangs.

Authorized licensed use limited to: Tsinghua University. Downloaded on August 14,2026 at 07:47:49 UTC from IEEE Xplore. Restrictions apply.

non-HLT instructions, or be halted by executing a HLT instruction during a system hang. The System Hang Detection module (SHD) provides a low-latency detection of, and recovery from, system 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 because the timeout interval for resetting the watchdog timer is usually a matter of seconds (to reduce false alarms, and the time overhead of the heartbeat).

OS kernel instrumentation, and watchdog timer modifications are required to reduce latency in detecting system hangs, and to avoid false 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., the operating system does not schedule processes), a system hang is declared.

The approach in this paper enables a 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 a system hang is that an operating system in a hang state does not relinquish the processor, and does not schedule any processes. Based on this fault model, system hang detection can be constructed as follows.

1) Use OS-level counters to separately track instructions executed by the application processes, and the operating system.
2) Observe periodically (e.g., on each context switch) whether the instruction count in each counter changes between the consecutive readings.
3) If the contents of the OS counter continue to change, and the counters associated with application processes are frozen, this is a clear indication of a system hang.

Two important issues are: i) to determine a time period during which to measure the dedicated OS-level, and application-level instruction counters; and ii) to have a mechanism to continuously, and accurately measure the number of instructions being executed in the selected time period.

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 system hangs, it may 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 typical processes in Linux 2.6, and 20/40/60ms in Windows XP), 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 cases, this bound is a large number due to the fact 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). To provide low-latency system hang detection, a much lower bound, based on profiling of system execution, can be applied 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.

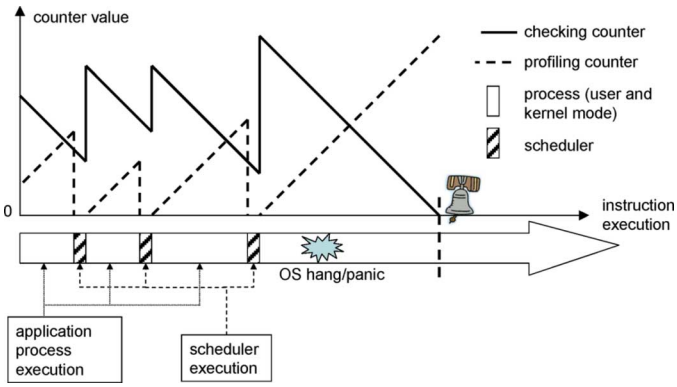


Fig. 6. System Hang Detection Using Instruction Counting.

The SHD module implements system hang detection using two counters:

- ***Profiling counter,*** which keeps track of the number of instructions executed in the current time slice by a process; and
- ***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 about 50 time-slices.

Fig. 6 illustrates the principle that the SHD applies for system 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.

The checking counter (the solid line) detects system 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 Fig. 6 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 system 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, the checking counter is not reset, and finally reaches zero (indicated as a bell alarm in Fig. 6). 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

Authorized licensed use limited to: Tsinghua University. Downloaded on August 14,2026 at 07:47:49 UTC from IEEE Xplore. Restrictions apply.

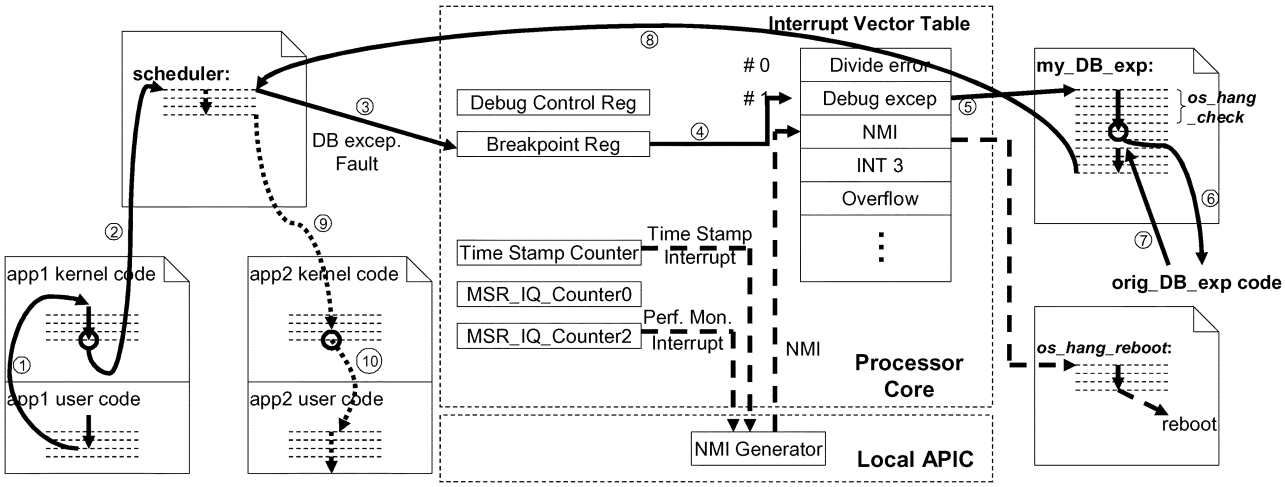


Fig. 7. Implementation of System Hang Detection in Linux 2.6.11 on a Pentium 4.

handler initiates system reboot.[8] The profiling-based checking makes the detection low-latency while adapting to changes in the system execution.

A. 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 due, for example, 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 a 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 a system hang when it reaches zero (forcing the system reboot). The fixed timeout value for 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.[9]

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.

[8]More advanced, sophisticated system recovery can be designed. For example, the erroneous device driver can be localized, uninstalled, and reinstalled. The involved application processes are recovered accordingly.

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

Priority-Based Scheduling. We assume that in a balanced system all tasks have the same priority. Although Linux, and Windows support 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.

B. SHD Implementation

Three pins, P_SCHL (scheduler interceptor), P_APIC (advanced programmable interrupt controller), and P_PMC (performance monitor counter), support the SHD module. To illustrate how the SHD works, the behavior of the module in Linux, as an example, is depicted in Fig. 7 (with implementation of the pins shown). The solid, and dotted lines illustrate the control flow of the processor execution. Circled numbers indicate the step order.

An application process, *app1*, enters the kernel mode to execute a system call, or to handle an interrupt (step 1 in Fig. 7). 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 typical 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).

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.,

Authorized licensed use limited to: Tsinghua University. Downloaded on August 14,2026 at 07:47:49 UTC from IEEE Xplore. Restrictions apply.

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[10] (step 8). The third task is required because the original debug exception handler by default disables the debug exception. After returning to the scheduler, the typical processing resumes (the next process is scheduled) (steps 9, 10). 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 a system 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 Fig. 7.

VI. Application Hang Detection Module (AHD)

Applications (or individual application threads) may hang due to incorrect function parameters, hardware faults, deadlock, or failed I/O. For example, it is reported in [36] that 6 out of 11,986 experiments with bad parameters lead to hangs of POSIX functions in Linux 2.0.18. 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].[11] 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 (e.g. in a form of nested loops) and code sections may overlap. 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 Fig. 8. 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.

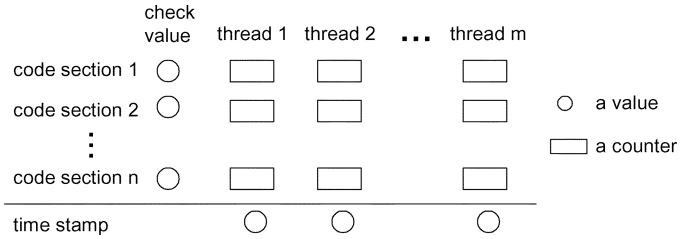


Fig. 8. The Counter Array Monitoring a Multithreaded Application in the AHD.

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.

A. 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 system hang detection, the fixed interval must be sufficiently large to avoid false alarms, e.g., 5 seconds.[12] Note that a typical 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, similar 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 insignificant 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.)

B. AHD Implementation

The AHD module employs services provided by four RMK pins: P_SCHL, P_SYSC (system call), P_PERI (periodical

[10]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.

[11]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.

[12]A thread scheduling may be delayed due to a heavy system load.

Authorized licensed use limited to: Tsinghua University. Downloaded on August 14,2026 at 07:47:49 UTC from IEEE Xplore. Restrictions apply.

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 Fig. 8. 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 Fig. 8, 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.*
  - Add the read number 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, indicate a hang.
  - Set the timestamp for the thread to the current system time.
- *Upon an instruction retirement:*
  - *The hardware counter increments by 1.*
- *On section_enter(section id) invocation:*
  - *Set the active bit for the section, and create related table entries if they are not present.*
  - Read the hardware counter's value c, and set 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:*
  - *Read the hardware counter's value, and adds it to the current count value of the section within the thread.*
  - Append the sum as a piece of count history in the path table for the section and the exit. Then compute the new check value[13] *according to the count history for the exit with the largest average instruction count.*
  - Clear the active bit, and removes unused table entries
- *Periodically perform the following check at a fixed interval:*
  - *Check the timestamps for all the threads of the process associating with at least one active section. If a timestamp expired, indicate a hang.*

VII. 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 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 schemes, e.g. Plank's libckpt [13], rely on user-level knowledge to decide when to take checkpoint for addressing consistency with the 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 as 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.

A. Generic Checkpointing

Each application process has its kernel state for correct execution of the process; and a transparent application checkpoint must address the consistency between the recovered application process, and the kernel state. For example, Fig. 9 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 Fig. 9) 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 the modified user memory pages, and the processor state. The TAC module eliminates the need to checkpoint all kernel states of the process by taking a checkpoint only when the kernel state is in a *safe point*, i.e., 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 IPC 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.

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

Authorized licensed use limited to: Tsinghua University. Downloaded on August 14,2026 at 07:47:49 UTC from IEEE Xplore. Restrictions apply.

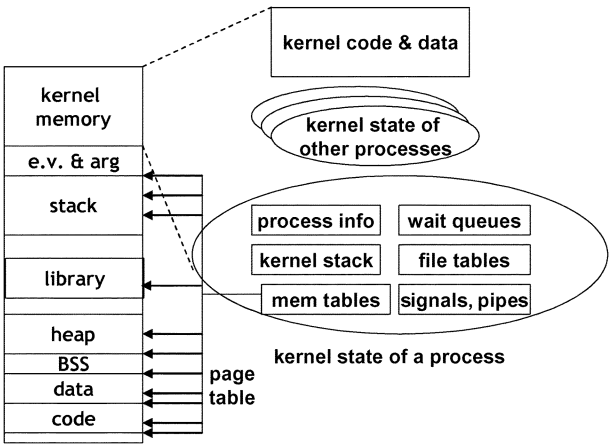


Fig. 9. Address space of a rocess in Linux.

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 states 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).

B. Custom Checkpointing for the Apache Web Server

When the application characteristics are taken into account, the generic checkpointing scheme can be customized to achieve better performance. We demonstrate this approach 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 Fig. 10(a) [25].

The parent server manages the pool of child servers through their process ID, 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. A state transition diagram[14] in Fig. 10(b) depicts the execution flow of an Apache child server process.

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, 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 (the 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. Because 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 Fig. 10(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 state 1, checkpoint 0 is restored, and the mutex is released if it is still held. When the application fails in states 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

[14]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.

Authorized licensed use limited to: Tsinghua University. Downloaded on August 14,2026 at 07:47:49 UTC from IEEE Xplore. Restrictions apply.

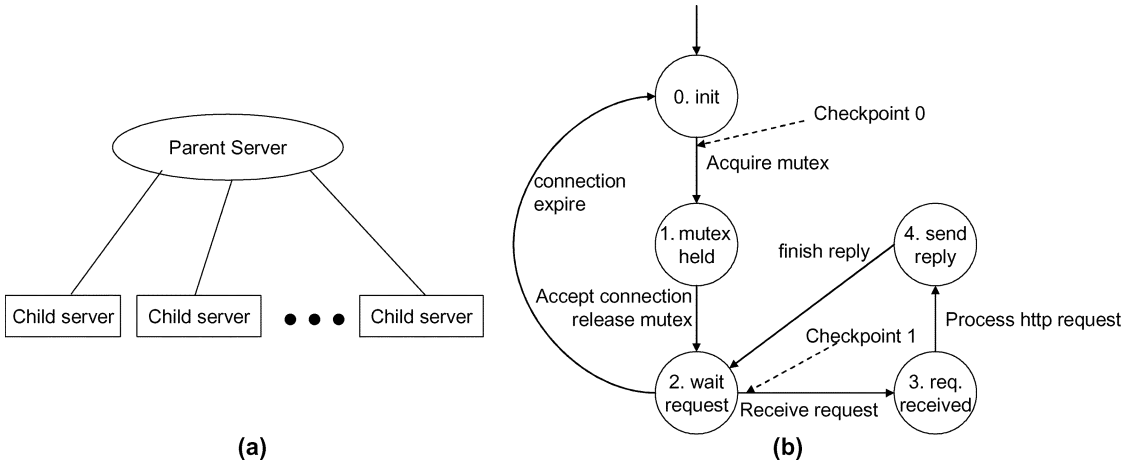


Fig. 10. (a) Process Architecture of Apache. (b) State Transition Diagram of the Child Server.

TABLE IV
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 system hangs without reporting any exception.

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.

VIII. EXPERIMENTAL EVALUATION

The RMK is implemented as a loadable kernel module in Linux 2.6.11, and as a device driver in Windows XP Professional SP2, on a Pentium 4 processor (1.5G Hz). The RMK pins are implemented as part of the RMK driver by taking advantage of the kernel functions, and variables exported from the operating systems, as well as the monitoring/debugging capability of the modern processors. Experiments are conducted on the two RMK implementations to evaluate its effectiveness, and performance overhead. 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. We mainly present the experiment results for the RMK on Linux here because the availability of Linux source code allows us to instrument the Linux kernel for in-depth observation of the system behavior during experiments.

A. Evaluation of System Hang Detection

Experiment Setup. To assess the effectiveness of the system hang detection, we need to create operational conditions that are likely to lead to system 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 system 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 IV.

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

Table V indicates that 9.0% of injected errors lead to system hangs, and that the SHD (System Hang Detection) module detects all hangs observed in our experiments. Table VI gives five examples of error propagations leading to system 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 can get corrupted due to the error propagation (*Global Descriptor Table (GDT)*, and *Task Stack Segment (TSS)* in the system are corrupted in examples 4, and 5). Surprisingly, kernel exceptions can be triggered repeatedly (e.g., spin_lock_unreleased is reported more than 30 times before GDT is corrupted in example 5).

SHD detects a hang after a predetermined number of instructions executed without a context switch, then reboots the system.

Authorized licensed use limited to: Tsinghua University. Downloaded on August 14,2026 at 07:47:49 UTC from IEEE Xplore. Restrictions apply.

TABLE V
ERROR INJECTIONS IN THE VICTIM DRIVER (1346 EXPERIMENTS) FOR LINUX

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**

TABLE VI
EXAMPLES OF ERROR PROPAGATIONS LEADING TO SYSTEM 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

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.

Experiment Results on Windows. Experiments with the same setup described above are conducted for the RMK implementation on Windows XP, and the results are reported in Table VII. Note that there is a difference between the terminology used in Table IV, and Table VII to define outcome categories. As we have access to Linux source code, we instrument the Linux system to study error propagation (see Table VI), and the terminology in Table IV is based on inside knowledge of kernel behavior (e.g. *kernel exception*). For the Windows system, analysis of error propagation is not possible due to the lack of the access to the Windows source code. Therefore, failure outcome categories are defined slightly different from those shown in Table IV (there is no *kernel exception*, and *system hang* replaces the *silent hang*).

Most of the experiments (59.4%) result in system crash (i.e. the Windows system reboots), 11.7% of the experiments result in fail silence violation, and system hangs are resulted in only 1.7% of the experiments. Compared with the experiment result in Linux (Table V), there are many more system crashes but fewer system hangs. This difference is because the default behavior of kernel exception handlers in Windows reboots the system, while the default behavior of kernel exception handlers in Linux allows the system to continue, which can lead to a system hang.

TABLE VII
ERROR INJECTIONS IN THE VICTIM DRIVER (802 EXPERIMENTS) FOR WINDOWS XP

Outcome Category	Number
Correct Output	218 (27.2%)
Fail Silence Violation	94 (11.7%)
System Crash	476 (59.4%)
System Hang	14 (1.7%)

B. Evaluation of Application Hang Detection

Experiment Setup. The web server Apache 2.0.55 on Linux is the target application for fault injection to evaluate the AHD (Application Hang Detection). A website consisting of hundreds of pages & files is set up on the target machine. A web client launches multiple requests from another machine. Because the application consists of tens of thousands of lines of code, fault injection is conducted into the main loop of the server (target code), which accepts web requests, and sends back replies (illustrated in Fig. 10(b)). Three code sections are identified in the target code: *initialization* (state 0, and transition to state 1 in Fig. 10(b)), *connection acceptance* (state 1, and transition 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 VIII, 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

Authorized licensed use limited to: Tsinghua University. Downloaded on August 14,2026 at 07:47:49 UTC from IEEE Xplore. Restrictions apply.

TABLE VIII
EXPERIMENT CAMPAIGNS FOR APPLICATION HANG DETECTION

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

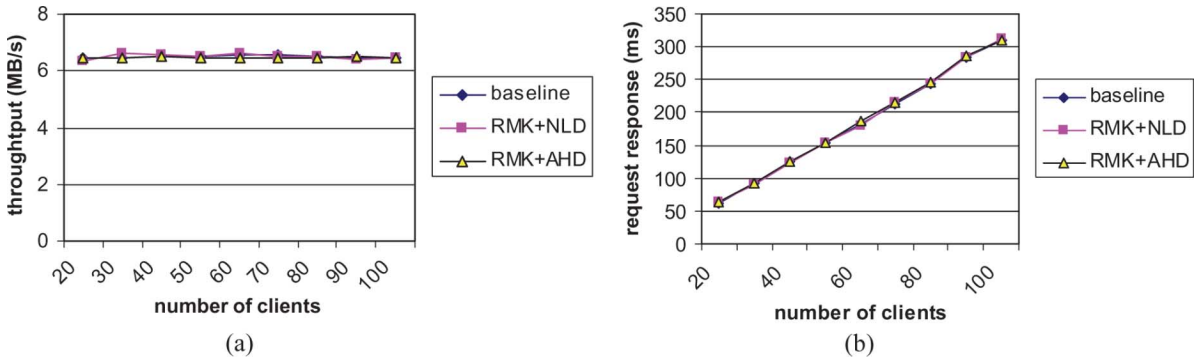


Fig. 11. Performance Overheads of RMK and SHD/AHD Modules. (a) Throughputs of the web server. (b) Average response time for a request.

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.

Bit flips are injected into the target code in Campaign 1. Table VIII 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 embedded in the application. The AHD module detects all hangs injected in our experiments.

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 many more instructions be executed within a code section (2,726,433 instructions are executed for the request for processing, while the largest instruction count for the previous 50 requests for processing is 449,493). The false positive may be avoided with a longer monitoring history (e.g. 100 requests), or a higher threshold.[15]

C. Performance Overhead of RMK, SHD, and AHD

Experiment Setup. Experiments are conducted to study the performance overhead of the RMK. Two 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 benchmark program 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 sets 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. Fig. 11 gives the server throughput (amount of data transmitted per second), and average response time for a request as a function of the 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 behavior is true 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.

Fig. 11 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 Fig. 11(a) is 6.51 ± 0.04 ms 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 of count values at context switches, and iii) periodic timestamp checks. Concerning the low frequencies of context switches, and timestamp checks, ii), and iii) incur negligible overheads. If code sections are entered & exited frequently, the overhead in i) is noticeable, but still small.

D. Performance Overhead of TAC

Three applications are used to evaluate the performance overhead of the generic checkpointing in TAC (Transparent Application Checkpoint) module: i) gzip, a SPEC2000 benchmark, ii) ns2, a popular network simulator, and iii) Apache, a

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

Authorized licensed use limited to: Tsinghua University. Downloaded on August 14,2026 at 07:47:49 UTC from IEEE Xplore. Restrictions apply.

TABLE IX
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

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. 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).

1) 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 IX summarizes the TAC performance overheads (the 95% confidence interval), and compares the results with the performance of the memory-dump approach.

Table IX 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 range 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 IX); 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

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

The measurement in the experiments shows $s = 24.55 \pm 2.02$ μs, and $r = 9.86 \pm 0.46$ μs (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 IX 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, as a process forking, and a process termination are included (for the Apache application, the overhead associated with the process forking and termination is 1.75±0.05 ms, much larger than the 0.019ms in TAC).

Comparison with Memory-Dump Checkpointing. The overall overheads for the three benchmark applications, 1.01 ms, 1.12 ms, and 1.29 ms, are less than 0.1% of the checkpoint interval (2 seconds), and justify the selection of checkpoint intervals of several seconds. Table IX (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 greater performance improvement than memory-dump checkpointing, especially for applications with large memory (1.29 ms versus 15.08 ms, a 91% improvement for ns2). This is because TAC effectively exploits the space locality of memory accesses performed by applications.

Authorized licensed use limited to: Tsinghua University. Downloaded on August 14,2026 at 07:47:49 UTC from IEEE Xplore. Restrictions apply.

TABLE X
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

2) Experiment Results for Custom Checkpointing for Apache: Experiments are conducted to evaluate the custom checkpointing for Apache with the support of TAC. The performances of three 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 X (with a 95% confidence level).

From Table X, one can see that, if there is no checkpointing provided for Apache, the recovery time is large (42.02 ms), which greatly degrades web service availability. The recovery times for memory-dump checkpointing, and custom Apache checkpointing, are similar (around 4 ms), 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).

IX. CONCLUSIONS

The paper describes the Reliability MicroKernel (RMK) framework, a loadable kernel module for providing application-aware reliability, and configuring reliability mechanisms. The RMK prototype has been implemented in both Linux, and Windows on a Pentium 4 processor; and supports the detection of application/system failures, and transparent application checkpointing. The experimental evaluation of the RMK using real-world applications shows that the system hang detection, and application hang detection, which exploit characteristics of application and system behavior, achieve high coverage, and low false-positive rates (1 out of 2000 experiments for application hang detection, and no false positive for system hang detection). Because the OS-level knowledge of applications/system is used, the RMK prototype has a low overhead in providing the transparent application checkpoint (less than 0.1% in the experiments), and application failure detection (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 system security policies on executed applications for detecting malicious attacks (e.g., a web server never launches a shell), and tracing system calls invoked within specific application functions to detect system anomalies.

REFERENCES

[1] D. Bovet and M. Cesati, "Clock and timer circuits," in *Understanding the Linux Kernel*, 3rd ed. : O'Reilly & Associates, Inc., 2005, ch. 6.1.

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

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

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

[5] M. Russinovich and Z. Segall, "Fault-tolerance for off-the-shelf applications and hardware," in *Proc. of the 25th International Symposium on Fault-Tolerant Computing*, 1995.

[6] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Julin, D. Orr, and R. Sanzi, "Mach: A foundation for open systems," in *Proc. 2nd Workshop Workstation Operating System*, Sept. 1989.

[7] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser, "Overview of the CHORUS distributed operating systems," in *USENIX Workshop on Microkernels and Other Kernel-Architectures*, Seattle, WA, 1992.

[8] N. Nakka, G. P. Saggese, Z. Kalbarczyk, and R. K. Iyer, "An architectural framework for detecting process hangs/crashes," in *The European Dependable Computing Conference 2005*, pp. 103–121.

[9] N. Nakka, Z. Kalbarczyk, R. K. Iyer, and J. Xu, "An architectural framework for providing reliability and security support," in *International Conference on Dependable Systems and Networks*, 2004, pp. 585–594.

[10] *"Operations Manual: PCI Watch-Dog Timer Card"* DECISION-COMPUTER Deutschland [Online]. Available: http://www.decision-computer.de/dowSHD/sonstiges/manualPCIwatchdog.PDF

[11] D. Beauregard, "Error injection-based failure profile of the IEEE 1394 bus," MS thesis, , 2003.

[12] Z. Kalbarczyk, R. K. Iyer, and L. Wang, "Application fault tolerance with armor middleware," *IEEE Internet Computing*, vol. 9, no. 2, pp. 28–37, 2005.

[13] J. S. Plank, M. Beck, G. Kingsley, and K. Li title, "Libckpt: Transparent checkpointing under unix," in *Conference Proceedings, Usenix Winter 1995 Technical Conference*, New Orleans, LA, January 1995, pp. 213–223.

[14] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The design and implementation of zap: A system for migrating computing environments," *ACM SIGOPS Operating Systems Review*, vol. 36, 2002.

[15] E. Pinheiro, "Truly transparent checkpointing of parallel applications," Dec. 1997 [Online]. Available: http://www.research.rutgers.edu/~edpin/epckpt/epckpt.ps.gz, Internal Report

[16] L. Wang, Z. Kalbarczyk, R. K. Iyer, H. Vora, and T. Chahande, "Checkpointing of control structures in main memory database systems," in *The International Conference on Dependable Systems and Networks*, 2004, pp. 687–692.

[17] D. Bovet and M. Cesati, "Checking the NMI watchdogs," in *Understanding the Linux Kernel*, 3rd ed. : O'Reilly & Associates, Inc., 2005, ch. 6.4.2.

[18] Y. S. Li, S. Malik, and A. Wolfe, "Performance estimation of embedded software with instruction cache modeling," *ACM Trans. on Design Automation of Electronic Systems*, vol. 4, no. 3.

[19] M. E. Russinovich and D. A. Solomon, *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, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *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.

Authorized licensed use limited to: Tsinghua University. Downloaded on August 14,2026 at 07:47:49 UTC from IEEE Xplore. Restrictions apply.

[24] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczyk, and R. K. Iyer, "Dependability assessment in distributed systems with lightweight fault injectors in NFTAPE," in *Proc. of the Dependable Computing for Critical Applications Conf.*, 1998.
[25] L. Stein and D. MacEachern, *Writing Apache Modules with Perl and C*. : O'Reilly & Associates, Inc., 1999.
[26] F. Salles, M. Rodriguez, J.-C. Fabre, and J. Arlat, "MetaKernels and fault containment wrappers," in *Proc. of the 29th International Symposium on Fault-Tolerant Computing*, 1999, pp. 22–29.
[27] J. Arlat, J.-C. Fabre, M. Rodriguez, and F. Salles, "Dependability of COTS Microkernel-based systems," *IEEE Trans. Computers*, vol. 51, no. 2, Feb. 2002.
[28] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, "Flashback: A lightweight extension for rollback and deterministic replay for software debugging," in *USENIX Annual Technical Conference, General Track*, 2004, pp. 29–44.
[29] P. Bernstein, "Sequoia: A fault-tolerant tightly coupled multiprocessor for transaction processing," *IEEE Computer*, Feb. 1988.
[30] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang, "Characterization of Linux Kernel behavior under errors," in *The International Conference on Dependable Systems and Networks*, 2003.
[31] Y. Huang and C. Kintala, "Software-implemented fault tolerance: technologies and experience," in *Proc. IEEE Fault-Tolerant Computing Symp.*, June 1993.
[32] N. Neves and W. K. Fuchs, "RENEW: A tool for fast and efficient implementation of checkpoint protocols," in *Proc. of the 28th International Symposium on Fault-Tolerant Computing (FTCS)*, June 1998.
[33] L. Wang, K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer, "Modeling coordinated checkpointing for large-scale supercomputers," in *International Conference on Dependable Systems and Networks*, 2005.
[34] M. Beck, J. S. Plank, and G. Kingsley, "Compiler-assisted checkpointing," Technical Report UT-CS-94-269, 1994.
[35] G. Hoglund and J. Butler, *Rootkits: Subverting the Windows Kernel*. : Addison-Wesley Professional, July 2005, ISBN: 0321294319.
[36] P. Koopman and J. DeVale, "The exception handling effectiveness of POSIX operating systems," *IEEE Trans. Software Engineering*, vol. 26, no. 9, 2000.

Long Wang is a Ph.D. candidate in the Department of Electrical and Computer Engineering at the University of Illinois, Urbana-Champaign. His research interests include fault tolerance, system reliability, and system analysis. Wang received an MS in computer science from the University of Illinois, Urbana-Champaign. He is a student member of the IEEE.

Zbigniew Kalbarczyk is principal research scientist at the University of Illinois, Urbana-Champaign. His research interests include automated design, implementation, and evaluation of dependable and secure computing systems. Kalbarczyk received a PhD in computer science from the Bulgarian Academy of Sciences. He is a member of the IEEE and the IEEE Computer Society.

Weining Gu is a Ph.D. candidate in the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign. He received an M.S. degree in electrical and computer engineering from the University of Illinois at Urbana-Champaign in 2004. He also received M.E. and B.E. degrees in computer science in Nanjing University of Aeronautics & Astronautics and Beijing University of Aeronautics & Astronautics in China, respectively. His research interests include the design of dependable/secure systems and networks, dependability evaluation, and benchmarking of computing and communication systems. He is a member of the IEEE.

Ravishankar K. Iyer is director of the Coordinated Science Laboratory, and George and Ann Fisher Distinguished Professor of Engineering at the University of Illinois, Urbana-Champaign. His research interests include reliable and secure computing, measurement and evaluation, and automated design. Iyer received a PhD from the University of Queensland, Australia. He is an IEEE fellow, an associate fellow of the American Institute for Aeronautics and Astronautics, and a member of the ACM, Sigma Xi, and the IFIP Technical Committee on Fault-Tolerant Computing (WG 10.4).

Authorized licensed use limited to: Tsinghua University. Downloaded on August 14,2026 at 07:47:49 UTC from IEEE Xplore. Restrictions apply.