appeared in Proc. European Conference on Computer Systems (EUROSYS), April 2010, pp. 181-194.
Won best student paper award.

Otherworld - Giving Applications a Chance to Survive OS Kernel Crashes

Alex Depoutovitch
Department of Computer Science
University of Toronto
depout@eecg.toronto.edu

Michael Stumm
Department of Electrical and Computer Engineering
University of Toronto
stumm@eecg.toronto.edu

Abstract

The default behavior of all commodity operating systems today is to restart the system when a critical error is encountered in the kernel. This terminates all running applications with an attendant loss of "work in progress" that is non-persistent.

Otherworld is a mechanism that microreboots the operating system kernel when a critical error is encountered in the kernel, and it does so without clobbering the state of the running applications. After the kernel microreboot, Otherworld attempts to resurrect the applications that were running at the time of failure. It does so by restoring the application memory spaces, open files and other resources. In the default case it then continues executing the processes from the point at which they were interrupted by the failure. Optionally, applications can have user-level recovery procedures registered with the kernel, in which case Otherworld passes control to these procedures after having restored their process state. Recovery procedures might check the integrity of application data and restore resources Otherworld was not able to restore.

We implemented Otherworld in Linux, but we believe that the technique can be applied to all commodity operating systems. In an extensive set of experiments on real-world applications (MySQL, Apache/PHP, Joe, vi), we show that Otherworld is capable of successfully microrebooting the kernel and restoring the applications in over 97% of the cases. In the default case, Otherworld adds zero overhead to normal execution. In an enhanced mode, Otherworld can provide extra application memory protection with overhead of between 4% and 12%.

EuroSys'10, April 13–16, 2010, Paris, France.

Categories and Subject Descriptors D.4.5 [*Operating Systems*]: Reliability – Fault-tolerance

General Terms Design, Reliability

Keywords Recovery, Kernel, Microreboot, Crash Kernel

1. Introduction

Computer systems periodically experience failures,[1] and this comes at a considerable cost. According to Patterson, organizations spend from 30 to 50 percent of computer system's total cost of ownership on preparing for and recovering from failures [26]. Software failures dominate hardware failures, and Patterson argues that faults in software are an unavoidable fact of our life that we have to cope with. While operating systems can ensure that an application failure does not affect the execution of other applications running on the same system, a fault inside the operating system kernel itself may result in a failure of the entire software system stack.

As shown by Chou et al., the number of bugs in operating systems increases with each release, since each new version tends to be more complex and tends to add more functionality [11]. According to the same study, the average time between when a bug is introduced to when a fix is released is 1.8 years for Linux kernels. Moreover, research suggests that the number of transient memory and processor failures is increasing [5]. These failures are intermittent and cause incorrect values to be read from memory or incorrect instruction results to be produced.

When a failure occurs inside the operating system kernel, it may affect any part of the software system and the recovery code itself may be affected by the failure. Most modern production operating systems take the simplified approach of simply rebooting the system when a failure is detected

[1] Throughout the paper, we follow the terminology established by Avizienis et al. [2]. A system *failure* or a *crash* occurs when the delivered service no longer complies with the agreed description of the system's expected function and/or service. An *error* is that part of the system state that is liable to lead to subsequent failure; an error affecting the service is an indication that a failure occurs or has occurred. The adjudged or hypothesized cause of an error is a *fault* or a *bug*.

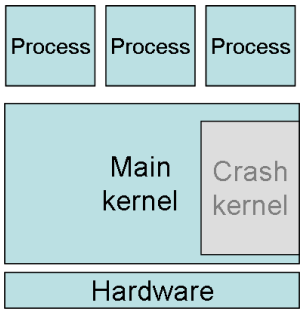

Figure 1. Main and crash kernels

and, as a consequence, the entire software stack fails and all volatile application state is lost. There have been attempts at creating operating system kernels capable of recovering from failures, such as Multics or MVS [1], but faults in these systems still resulted in outages and reboots in more than 19% of the cases [28], and half of the operating system code consisted of error recovery code [31].

The key idea behind our work is that an operating system kernel is simply a component of a larger software system, which is logically well isolated from other components, such as applications, and therefore it should be possible to reboot the kernel without terminating everything else running on the same system. Following Candea et al. [8], we call such a reboot a *kernel microreboot* to distinguish it from a full system reboot. Microrebooting a component as important as the operating system kernel may be difficult without support from some of the applications. But, even for such applications, we will argue that this is possible with minimal and straightforward changes to application code. The solution we propose, called "Otherworld", does not add to hardware costs, preserves the latest state of the application, and can be applied to production operating systems. Our solution can operate in two modes: in simple mode with no run time overhead that does not protect from error propagation or in memory protected mode that provides a higher level of protection against error propagation at the cost of between 4% and 12% overhead.

Two properties of existing operating system kernels complicate the process of microrebooting the kernel without affecting running applications. First, the kernel resides in a privileged layer underneath all applications, so there is no other software component that can manage kernel reinitialization without destroying all applications running above the kernel. Secondly, the kernel itself contains data critical for running applications, such as a physical memory page tables, location of data paged to disk, as well as state belonging to opened files and network connections.

To address these issues, we have designed a mechanism called "Otherworld". We propose having two operating system kernels resident in physical memory. The first (*main*) kernel performs all activities an operating system is responsible for. The second (*crash*) kernel is passive and is activated only when the main kernel experiences a critical failure (Fig. 1). If the main kernel crashes (i.e., "panics"), instead of rebooting, it passes control to the crash kernel. The crash kernel is not affected by the error because it has been passive and is protected by standard memory hardware. After obtaining control, the crash kernel initializes itself using only the limited region of memory reserved for this purpose by the main kernel. All information on running applications in the main kernel as well as the application data still exists in memory and is accessible to the crash kernel. This allows the crash kernel to reconstruct the state of each application and pass control to the applications without losing data. We refer to this reconstruction process as application *resurrection*.

An application can optionally register a special user-level function, called *crash procedure*. This function is called by the crash kernel notifying the application that a kernel microreboot has occurred and that the application has been resurrected. The crash procedure is called similarly to the way application exception (signal) handlers are called. The crash procedure can either save application state to persistent storage and restart itself, or it can restore relevant parts of the system state that are not automatically restored by the crash kernel and instruct the crash kernel to continue application execution from the point at which execution was interrupted by the kernel failure.

We evaluated the viability of our mechanism with different classes of applications: interactive applications, a database server, a web application server, and a checkpointing solution. We were able to successfully resurrect each of those applications after kernel restarts caused by kernel failures in more than 97% of the cases. For the applications we considered, we found that writing a suitable crash procedure was either not necessary or only a matter of writing few dozen lines of code and did not require a deep understanding of applications internals.

In the next section we discuss related work. Section 3 describes the architecture and implementation details of Otherworld. The reliability of Otherworld is discussed in Section 4. Section 5 gives examples of crash procedures for different applications and discusses details of their implementation. In Section 6, we experimentally evaluate our approach and conclude in Section 7.

2. Related Work

A computer system can have several distinct sources of failure, and different techniques are used to address each of them. Uninterruptible power supply units allow computer systems to tolerate power outages. Continuous replication, implemented in products like VMWare Fault Tolerance [32], protects a computer system from hardware failures. Operating systems protect applications from faults in other applications. Orthogonally, Otherworld allows computer systems to

tolerate faults within the operating system itself. There are other fault tolerant techniques that implement similar functionality that we describe below, but Otherworld has some advantages over each of them. Otherworld is designed to run on conventional hardware without requiring redundant hardware. It does not put any restrictions on the operating system architecture or programming language used. It introduces no or minimal runtime overhead (based on mode of operation).

One of the more popular techniques that allow application data to survive operating system failures is periodic checkpointing. However, periodic checkpointing introduces overhead that is significant for applications that use a large amount of frequently changing data. Laadan and Nieh show that checkpointing a MySQL server serving one client takes more than a second during which time the system is unresponsive (even on a system with Fibre Channel hard drives) [20]. King et al. measured that checkpointing a virtual machine with 256MB of memory taken every 10 seconds adds 15-27% to the benchmark execution time [19]. In-memory checkpointing reduces overhead by an order of magnitude [27] but does not protect against operating system failures and effectively halves the amount of available physical memory. Another problem with checkpointing is that application state restored from the checkpoint is typically outdated, which may result in data loss. For example, transactional changes, which a client may consider to have been committed to a database, may be rolled back as a result of restoring data from the checkpoint.

In contrast to checkpointing that periodically captures a snapshot, Otherworld effectively captures a snapshot at the time of the kernel failure. It thus does not have the overhead and data loss problems that checkpointing has. On the other hand, checkpointing has several advantages over Otherworld, such as protection against power failures and application failures. Otherworld can also complement checkpointing: if the user is relying on in-memory checkpointing for protection against application faults, Otherworld can be used to protect the in-memory checkpoints against operating system crashes.

Microkernel-based operating systems provide improved fault isolation by locating some operating system components in separate address spaces. This makes it possible to restart individual components when they fail. For example, Minix has a reincarnation server that can restart other operating system components and drivers [17]. Designers of the CuriOS operating system propose that operating system components store part of their state in their client's address spaces and access it using the interface provided by the kernel [12]. On a failure, the failed component is restarted and obtains access to the client state stored by its previous instance. The performance overhead of this approach is still to be evaluated. The authors showed that their microkernel design does not necessarily protect from an error in one component propagating into other components; experiments determined that in 6% of the cases, the operating system ultimately crashes because of error propagation.

Rebooting the entire kernel instead of separate components makes Otherworld applicable to existing commercial operating systems (such as Linux, Windows, MacOS), which tend to be monolithic. Another advantage of Otherworld is that it is capable of operating with no runtime overhead and guarantees that errors in the main kernel do not propagate to the crash kernel. Minix and CuriOS are not protected from faults inside the microkernel, reincarnation server, or in any of the components it depends on, e.g., file system driver. Our approach provides protection against failures in any part of the operating system, except for about 100 lines of code that transfers control from the main kernel to the crash kernel.

Swift et al. improve the fault tolerance of existing monolithic operating system kernels with the *Nooks* framework, but only target device drivers faults [29, 30]. Nooks improves driver isolation from the rest of the kernel using memory protection and run-time type validation of API function calls arguments, significantly improving failure detection and reducing failure propagation. However, the performance overhead of Nooks varies from 3% to 100% depending on the workload.

Peter Chen et al. proposed Rio, a reliable file write-back cache, whose contents is preserved across system reboots [25]. Other research groups have also investigated ways of preserving specially designated regions of memory across operating system crashes and subsequent reboots [4, 7]. These regions of memory can only be used for saving application state, so the choice of their size is a trade-off between use of physical memory and the amount of state that can be saved by applications. Otherworld has the advantage that it preserves the entire application state. Moreover, the application does not need to regularly update the protected region with the latest, most critical data, thus avoiding run-time overhead, estimated by Baker et al. to be around 5%[4].

The MVS operating system contained routines for recovery from kernel failures [1]. The weak point of the MVS design is that the recovery routine executes in the kernel, which may have been already damaged by the fault. MVS did not provide a recovery routine for 34% of errors observed in the field, and even when recovery routine was provided, the recovery failed in 44% of the cases [28].

Yang Chen et al. designed and implemented kernel microrebooting in TinyOS, a special-purpose sensornet operating system designed to run on microcontrollers with tens of kilobytes of RAM [10]. To support kernel microrebooting, they rely on programmers explicitly specifying in the program code the state to be preserved during the reboot and compile the program and the kernel with a specialized compiler. Otherworld is designed to reboot much more complex general-purpose operating system kernels, such as Linux. It does not require a special compiler and in many cases does

not require changes to the application source code or even recompiling the application.

Goyal et al. developed KDump - a mechanism that enables booting into a new kernel while preserving the physical memory contents of a crashed system [14]. KDump's new kernel is used only to create a physical memory dump for further investigation - there is no attempt to recover applications. Our work takes this idea a step further by restoring all application memory state and attempting to continue application execution.

3. Design and Implementation

Our proposed technique of microrebooting a kernel to allow applications to survive kernel crashes with high probability is shown in Figures 2 - 5. The Otherworld architecture consists of three parts. First, there is a modified operating system kernel that microreboots itself without destroying running applications. Second, there is code inside the crash kernel that resurrects application processes after a microreboot. Third, for each application there is an optional *crash procedure*, an application-defined, user-level function called by the crash kernel on resurrection. The system operates in five stages:

1. At first boot, the system configures itself and loads the crash kernel image into a region of physical memory reserved for the crash kernel.
2. The system runs normally under the control of the main kernel. At any time, applications may register a crash procedure with the kernel.
3. On a kernel failure, control is passed to the crash kernel, and the crash kernel initializes itself within the memory reserved for it.
4. After initialization, the crash kernel resurrects the applications and calls their crash procedures, if registered.
5. The crash kernel takes control over all remaining system resources, morphs itself into the main kernel, and installs a new crash kernel.

We implemented our mechanism in Linux, but our architecture is generic and can be applied to other operating systems as well. We now describe each of the stages in more detail.

3.1 Setup

Initially, the computer system is booted normally by loading and initializing the main operating system kernel. In order for the system to be capable of performing a microreboot, the main kernel reserves a special region of physical memory (e.g., 64 MB) for a crash kernel. The crash kernel image is loaded into this region and is left there untouched and uninitialized, protected by memory hardware. As long as the main kernel operates without a failure, the crash kernel image is left intact in this region of physical memory, and its code is never executed. In our implementation under Linux, we use the KDump mechanism to load the crash kernel into memory and pass control to it after a failure has been detected [14].

Any user application that wishes to be notified after a kernel microreboot can register a crash procedure. The address of this procedure is stored in the process descriptor of the main kernel and serves as an entry point to be called if and when the crash kernel gains control.

In order to simplify resurrection and reduce the number of main kernel data structures that we have to retrieve and rely on, we modified the main kernel so that the information necessary to recreate resources that belong to the applications is easier to access. For example, in the Linux kernel, information relating to open files is located in the *file*, *inode*, and multiple *dentry* structures, but in order to recreate the open *file* structure for the file, only the file location, name, open flags and current file offset are required. We keep the location, name, and open flags specified by the application during the *open* call, in the *file* structure where the file offset is also stored. As a result, we only need to rely on one structure to recreate the kernel open file state.

In our current implementation, we use the same kernel source to build both the main kernel and the crash kernel. Although we use two different disk files for the main and the crash kernels, the only difference between these kernels is the initial kernel memory offset. Common source code for both kernels has advantages and disadvantages. One advantage is that it is easier for the crash kernel to access the main kernel data structures since they both use the same structure layout. In addition, modifications to one of the kernels are automatically applied to the other, as both kernels are built simultaneously. On the other hand, there are no impediments to having the crash kernel be different from the main kernel. Although this approach is more complex to implement, it has one important advantage. If the kernel fault that triggered a failure is not intermittent, e.g., was caused by some particular combination of system call arguments, resurrection of the application that triggered the fault could cause the same fault to be triggered again, since the application will retry the system call when run under the crash kernel. Using different kernel versions would allow us to successfully recover from this situation.

3.2 Response to Kernel Failure

When the main kernel experiences a critical error, instead of rebooting, it issues non-maskable interrupts to all processors except the one that executed the code causing the failure. Upon receiving a non-maskable interrupt, each processor saves the hardware context of the thread that it was executing, then sets a global flag indicating that it has saved its context, and brings itself to a halt. This ensures that CPU registers are saved on the corresponding kernel stack for all user threads by the time control is passed to the crash kernel. Later, it will allow the crash kernel to retrieve this context and continue thread execution similar to the way a regular

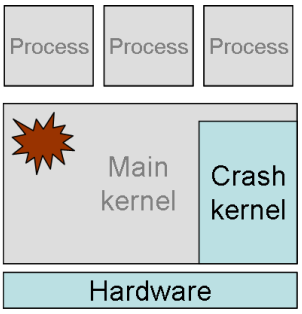

Figure 2. Failure is detected in the main kernel

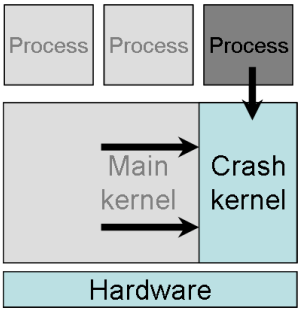

Figure 3. Crash kernel retrieves information from the main kernel

context switch occurs. The processor that was executing the code causing the failure waits for all other processors to halt, removes the memory protection from the crash kernel image, and jumps to the initialization point of the crash kernel. From this moment on, no main kernel code is executed, and the crash kernel controls the system.

The crash kernel initializes itself normally with the exception that it only uses the memory region originally reserved by the main kernel for this purpose (Fig. 2). Although, the crash kernel uses only the reserved region of memory during the resurrection, after the resurrection process is complete, the crash kernel starts using all the physical memory installed on the system (as described in Section 3.6). In order to be able to dynamically change the amount of physical memory available, the startup code of the crash kernel has to allocate extra page descriptors that are not used by the crash kernel during the resurrection process but will be used when the resurrection process is complete. This is the only change to the stock Linux startup code that we had to make for the crash kernel.

In order not to corrupt any pages that were swapped out by the main kernel, we use two swap partitions in our system: one is used by the main kernel and the other by the crash kernel. Initialization scripts can query the kernel to determine if it is the main or the crash kernel. Based on which kernel is booting, the initialization scripts choose the appropriate partition. The crash kernel and the main kernel share the same initialization scripts, load the same device drivers,[2] and mount the same file systems at the same mount points. As a result, the application environment of the crash kernel is the same as that of the main kernel, which makes kernel reboots more transparent to applications and simplifies the writing of crash procedures.

3.3 Application Resurrection

After the crash kernel completes its initialization, it starts a recovery phase in which it accesses the kernel structures of the main kernel in order to resurrect applications (Fig. 3). We believe that in most scenarios, the end user is interested in resurrecting only a few important processes, e.g., the database server, the web server, the text editor, etc. The other processes, such as the window manager, the mouse server, or the cron daemon, do not hold important state and can be safely restarted without resurrection in most scenarios. This completely eliminates the possibility of any side effects on these processes that might have been caused by the kernel crash failure. After the crash kernel finishes its initialization, the interactive user is presented with a list of the processes that were running on the system at the time of the crash. The user can then select the processes that should be resurrected. Alternatively, a resurrection configuration file that identifies which processes are to be resurrected automatically can be specified. The startup script consults this file to determine which processes to resurrect. The latter option is intended to be used by server systems for autonomic recovery, and we used it during our automated fault injection experiments (Section 6).

The first step of the resurrection is to recover process descriptors. In Linux, the process descriptors are organized in a linked list. The location of the first element of this list is stored in a global variable in the kernel. The starting physical address of the kernel is constant and configurable at kernel compilation time and, therefore, the crash kernel is aware of the physical address of the first element of the process list.

Next, the crash kernel retrieves the swap area descriptors from the main kernel, stored in a fixed size array accessible through another global variable. Each descriptor describes one swap partition and contains a pointer to the *file* structure that corresponds to a regular file or a device file that stores the swap area. Since the symbolic name of the device is stored in this structure, the crash kernel can reopen it.

For each process that is to be resurrected, the crash kernel creates a new process. The kernel portion of the virtual address space of the newly created process is the same as for any other process running on the crash kernel. The user

[2] Both the crash and the main kernel load the same set of drivers. Note that, we do not modify any drivers, and drivers initialize the devices as during the initial system boot. If a driver developer wants to use, as an optimization, different logic to re-initialize a device after a crash occurred, it is possible for the driver to ask the kernel if it is a crash kernel and even access the memory of the main kernel through the functions provided by the crash kernel.

portion of the virtual memory space of the newly created process is created to be a copy of the user portion of virtual address space of the process being resurrected. To do this, the crash kernel obtains the list of memory region descriptors of the process being resurrected from the main kernel memory. For each memory descriptor in the list, the crash kernel creates a new memory descriptor with the same attributes. Some of the memory regions may have been mapped to a disk file. In this case, there is a pointer to the corresponding *file* structure. These files are reopened by the crash kernel and mapped to the corresponding memory region.

The next step is to retrieve the contents of each virtual memory page within the memory region. For each page, the crash kernel retrieves from the main kernel memory the corresponding entry of the hardware page table of the process being resurrected. If the entry references a physical memory page, a new page is allocated in the crash kernel and the content from the corresponding page of the main kernel is copied into it.[3] For each entry that corresponds to a page that was swapped out to disk by the main kernel, a new page is allocated in the crash kernel's swap partition (which, as we mentioned earlier, is different from the main kernel swap partition). The contents of the newly allocated page is copied from the corresponding page of the main kernel swap partition. This fully restores the user-level memory space of each target process.

After the application memory space of a process is restored, the crash kernel restores the files that were open for the process. The process descriptor structure from the main kernel contains a pointer to a file descriptor table (*files_struct*), which describes all files that were open for the process when the main kernel failed. The crash kernel reads the name, location, open flags, and current offset from that table and reopens the files accordingly. In order to make the reopening of the files transparent to the application, the crash kernel places each reopened file into the same position within the file table as it was in the main kernel and restores the current offsets. The last step of resurrecting a process's open file is the flushing of its dirty file buffers. File buffers are organized in a tree. The root of this tree is accessible through the file descriptor. Each leaf element of the tree contains a pointer to a descriptor of a physical page with file data. Page descriptors contain the *dirty* flag, which has been set by the main kernel to indicate that the corresponding page needs to be saved to disk, and the offset of the data relative to the start of the file. The crash kernel saves all pages with a set dirty flag to disk.

With respect to terminals, the current Otherworld implementation can only restore the state of physical terminals. In order to do this, the crash kernel checks the type of the terminal attached to the process being resurrected, and if it is a physical terminal, its state is restored as a part of the application resurrection process. The screen contents of the physical terminal in Linux is stored in a kernel buffer, which can be indirectly obtained from the process descriptor. The crash kernel opens a new terminal and sets the terminal settings and the screen context from the terminal that the application used at the time of the main kernel failure.

[3] As an optimization, one can directly map the physical page instead of copying it, which would significantly increase the speed of resurrection of large processes.

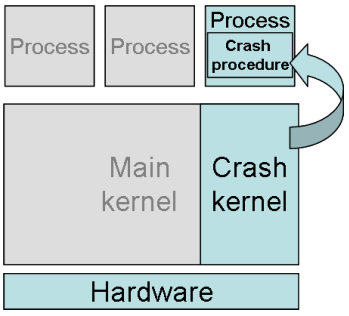

Figure 4. Application is resurrected and its crash procedure is called

At this point, the crash kernel is ready to start executing the resurrected process. If the process has registered a crash procedure, the kernel allocates a temporary stack in the user space and calls the crash procedure. This provides the resurrected process the opportunity to execute recovery code with all of the process's global data available (Fig. 4). Depending on the value returned by the crash procedure, the crash kernel will either terminate the process or continue execution of the process from the point at which it was interrupted by the crash.

In the current prototype, in addition to the application memory space, the crash kernel also resurrects all open and memory mapped files, signal handler descriptors, shared memory, and physical terminals. We have not yet implemented the resurrection of the various IPC resources, such as sockets, pipes, or pseudo terminals. Application crash procedures have to be added to programs that use these resource types in order to restore them in an application-specific manner or in order to at least shutdown the application gracefully after having saved application state to persistent storage. But, as we will show in Section 5, even our limited prototype is applicable to a wide range of applications. While this is a serious limitation (requiring crash procedures to reestablish these IPC channels), we believe these resources are resurrectable, albeit non-trivially, and plan on working on them next. Nearly all network protocols in Linux use sockets. However, the data associated with each socket will differ, depending on the protocol for which the socket was opened. As a result, the resurrection of sockets will also have to be protocol dependent. Most applications use TCP over IP or UDP over IP. Because IP and UDP do not guarantee packet delivery, it is safe to discard any payload data during resurrection, and only connection parameters associated with a

	Crash procedure defined	No crash procedure defined
All resources were resurrected	The crash procedure will be called. It can either save data to disk and restart the application or instruct the crash kernel to continue the execution of the application.	The crash kernel will continue the execution of the application.
Some resources could not be resurrected	The crash procedure will be called. The crash procedure can either restore resources itself and continue execution or save application state and restart the application.	The resurrection will fail.

Table 1. Interactions between the crash kernel and the application being resurrected.

socket needs to be resurrected, including IP address, socket options, and datagram size. For TCP, additional connection parameters need to be resurrected, including the current sequence number and window size. Moreover, all outbound payload data as well as acknowledged inbound payload data needs to be resurrected. We expect that the finer details of the TCP/IP implementation, large quantity of parameters associated with a TCP/IP stack, and the requirement that resurrection has to be completely transparent to the remote host will make resurrecting TCP/IP sockets non-trivial. Moreover, the resurrection time has to be smaller than a network timeout.

We believe that resources for local IPC structures are also resurrectable. For example, pipes are implemented as a circular buffer of memory shared between two or more processes. All accesses to pipe data structures are serialized using a semaphore, and when the semaphore of a pipe structure is not locked, then the structure should be in a consistent state. If the semaphore of a pipe structure is locked, then the structure was being accessed when the kernel failed and one must assume that the structure is in an inconsistent state preventing resurrection. The amount of time shared IPC data structures are locked depends on application usage; e.g., if two processes start exchanging a lot of data through the pipe, the pipe may be in an inconsistent state for a noticeable amount of time. As a result, we expect that the resurrection of IPC resources resurrection may fail at a higher rate compared to that of private process resources, such as process or memory descriptors.

3.4 Crash Procedure

After resurrecting all resources, the crash kernel calls the crash procedure if the process has registered one. Crash procedures serve several purposes. First, they are used to detect potential data corruption in an application-specific way. Since a kernel failure is an infrequent event, the application can afford to do elaborate data consistency checks. The problem of detecting data corruption is complex and interesting by itself, and we will leave it for future work.

Second, crash procedures are used for application-specific resource resurrection. The more resources being resurrected by the crash kernel, the higher the probability of encountering data corruption, where the crash kernel may not be able to restore all of the application resources. Also, some resource types can not be resurrected due to implementation limitations, as was outlined in Section 3.3. The crash kernel reports to the crash procedure which resource types it could not resurrect, by providing a bitmask argument to the crash procedure. Each bit in the bitmask corresponds to a resource type (e.g., network sockets or pipes) that the crash kernel did not restore due to implementation limitations or data corruption. An advanced crash procedure can resurrect these resources using application-specific logic, for example reopening network connections.

Finally, the crash procedure must determine whether it wishes to continue executing as is (e.g., if all resources were successfully resurrected), whether it wishes to save a portion of its state to persistent storage and restart, or whether it deems the restoration to be unsuccessful and gives up by exiting.

While an application without a crash procedure often can be resurrected, having a crash procedure that saves application data to the persistent storage and restarts the application, increases the probability of successful resurrection. As shown by Chandra and Chen, application-specific recovery is much less likely to be affected by faults within the operating system than application generic recovery [9]. In many cases (e.g., for non-critical interactive applications) continuing the execution so as to minimize inconvenience to the user at the expense of a slightly higher risk of data corruption is acceptable. For more mission critical applications, like databases, one may prefer to always save application data to the persistent storage and restart the application to eliminate all potential side-effects of the fault.

3.5 Resurrection Levels

Depending on the system resource types consumed by the process and the application's own preferences, different levels of resurrection are available, as summarized in Table 1. If a process did not register a crash procedure before the crash, and the process only uses resources resurrected by the crash kernel, then the crash kernel can continue the execution of the process from the point where it was interrupted by the crash. If, at the time of the crash, the process was executing in user mode (on a processor different from the one that experienced the failure), execution of the process will continue with the next instruction and the crash goes completely unnoticed by the process. If the process was in the middle of

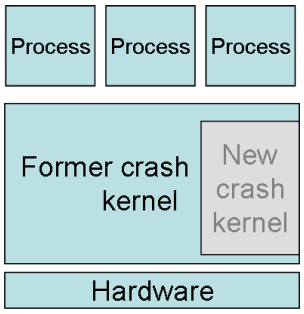

Figure 5. The crash kernel takes over the system and morphs into the main kernel

a system call, then the crash kernel will abort the call with a return error code that signals to the process to retry the call. Some applications may have to be modified to handle this error code correctly by re-executing the system call. If no crash procedure was registered before the crash, and the crash kernel was not able to resurrect all of the kernel resources consumed by the application, the resurrection will fail.

3.6 Morphing into a Main Kernel

After a kernel failure, it is important not only to restore application state but to also restore the full functionality of the system and protect the system from failures that may occur in the future. After all target application processes are resurrected, the physical memory that belonged to the main kernel is no longer needed. The crash kernel thus reclaims all of the available physical memory and adds it to its free memory list. One region of the reclaimed memory is reserved, and another crash kernel is loaded into this region. As soon as this is done, the crash kernel starts playing the role of the main kernel, and the newly loaded kernel becomes the crash kernel. As a result, the system is running with a fresh kernel, which is free of state corruption caused by the fault, the applications that were running at the time of failure were able to preserve their data and possibly continue execution, and the system is again protected from the failures (Fig. 5).

3.7 Kernel modifications

We have implemented Otherworld in Linux kernel version 2.6.18. The implementation required changing 400 lines of the existing code and adding 2,300 new lines of code. Modifications to the startup code, the file management code, and to the *clone()* system call were necessary. The startup code modifications were required to be able to add memory to the crash kernel after the resurrection process is complete and to reserve the region in which the new crash kernel may be loaded. File management code had to be modified in order to simplify the restoration of open files. Both process resurrection and the cloning of an existing process have a lot in common, since in both cases the copy of another process is created. Because of this, we modified the existing *clone()* call to handle both operations. Most of the new code was added for retrieving and recreating process information from the failed main kernel.

4. Data Corruption

The practicality of microrebooting an operating system kernel depends to a large extent on the probability that the bug that caused the kernel to crash also corrupted application memory and/or kernel structures needed for recovery.

Most faults in the operating system kernel conform to the fail-stop model and cause an immediate crash, leaving application data intact [3, 15, 22, 28]. However, there are some faults that do not result in an immediate operating system crash, thus potentially leading to data corruption. Application data can be corrupted *(i)* when kernel data structures that describe kernel resources owned by application are corrupted or *(ii)* when application data itself is corrupted.

In our implementation, as we will show in the evaluation section, structures required to resurrect a process occupy less than 0.12% percent of the total virtual address space size even on 32-bit platforms. The code that manipulates these structures in Linux constitutes approximately 2% of the total Linux code size. Moreover, the error rate of the process and memory management code relative to the code size is more than 3 times lower than that of the other parts of the kernel [11]. This leads us to expect the probability of corruption of these data structures to be low. We attempt to confirm this hypothesis later in the evaluation section.

The probability of undetected kernel data corruption can be significantly reduced by several simple, but effective, techniques. First, much of the state in the kernel is already duplicated in order to speed-up operations. For example, memory page information in Linux is stored in hardware page tables and in Linux *page* memory structures. Protection bits of each memory page table entry are duplicated in memory region descriptors. Kernel structure corruption can often be detected by carefully checking data integrity using appropriate rules. This type of analysis is only necessary after a failure and thus does not add overhead to the normal operation of the system. Secondly, one could add checksums or data duplication to the most important data structures, such as process descriptors and memory maps. This would introduce some run-time overhead but would ensure that corruption will not go undetected.

Another concern is that a kernel bug may have corrupted the application memory space before crashing the operating system. As was shown by Chandra and Chen, if we only consider the memory regions that contain data important for saving application state, the probability of application data corruption caused by faults within the operating system kernel is 1%-4% [9]. This probability can be further reduced by protecting the user memory space using standard hardware features. In a set of experiments, we protected the user por-

Application	Crash procedure	Modified lines of code
vi	Not required	0
JOE	Not required	1
MySQL	Required	75
Apache	Required	115
BLCR	Not required	0

Table 2. Modifications to the applications to support Otherworld.

tion of the memory space every time the application makes a system call to lower the probability of kernel faults corrupting application data. We used a separate set of page tables that maps only the kernel portion of the address space but not the user portion, and we switched to this page set on every system call. Any attempt by the kernel to directly access the user portion of the address space would then result in a kernel failure. The kernel is only able to read or write to the user space through specially designated functions that switched to the page tables that had the user portion of the address space mapped, perform necessary operation, and switch the page tables back. As we will show in the evaluation section, the cost of this protection is a few percent of overhead.

While a kernel microreboot may not be able to guarantee protection from memory corruption errors, it should be noted that alternative techniques also cannot not provide such guarantees. For example, an error in the kernel may corrupt application data before a checkpoint is taken, corrupting the checkpoint as well [22].

5. Case Studies

In this section we describe the benefits that Otherworld can provide for certain types of applications and estimate how difficult it is to write crash procedures for these applications if they are needed at all. We show that even simple crash procedures are able to restore application state of many real-world applications and improve application fault tolerance. We tested Otherworld with 5 different applications: the vi and JOE text editors, the MySQL database server, the Apache/PHP bundle, and the BLCR checkpointing solution. The results are summarized in Table 2.

In some cases, resurrection required a crash procedure. We found that writing a simple crash procedure does not require a deep understanding of application internal details. Applications that we considered have functions that serialize and deserialize important state to and from a file or a byte stream. The task of writing a simple crash procedure then includes:

1. identifying such functions,
2. adding a crash procedure that calls the serialization function to save application's state to the persistent storage and restarts the application,
3. modifying the application startup code to call the deserialization function supplying it with the application state that was stored during the previous step.

Internals of the functions used or the internal format of the data they produce is not important for the purpose of writing a crash procedure. We were able to easily identify and use such functions in 1-2 days for Apache/PHP and MySql without any prior knowledge of their internal design. We describe each application we considered in the following subsections.

5.1 Interactive Applications

For interactive applications, such as text and graphic editors or computer games, losing state due to a kernel crash results in all work since the last save operation being lost. While text editors typically autosave every few minutes, many other programs do not have this feature (e.g., graphic editors such as Photoshop or GIMP), in part because the size of the image being edited or game being played may be tens or even hundreds of megabytes and saving this much data has a significant performance impact. Moreover, we are not aware of any text editor that saves additional application state, such as the undo data, as a part of autosave. For such applications, Otherworld's ability to continue application execution after a kernel failure offers significant advantages.

A text editor is a good example of an application that can be resurrected without having to be modified. We tested Otherworld with two popular text editors: vi and JOE, which are shipped as part of many Linux distributions. JOE, in particular, contains a lot of advanced functionality, such as support for multiple windows, macro execution and syntax highlighting. Vi did not require any modifications in order to be resurrected, and no document data or application state was lost across kernel failures during our testing. Initially, JOE failed after resurrection because it treated any error code returned by the console read function as a critical error and terminated itself. Changing one line of code to reissue failed console reads allowed JOE to be resurrected without any other modifications, making any kernel crash transparent to the user. In both cases, the applications were able to run across a kernel failure without requiring a crash procedure. After resurrection, the user was presented not only with the latest contents of all documents, but also with the undo buffer, relative window positions and other application state preserved.

5.2 Databases

Another important class of application that can benefit from Otherworld are database management systems. Storing data in RAM instead of disk can improve server performance by up to 140 times [21, 24]. However, a key reason why in-memory databases are problematic is that data is lost when the operating system crashes. Otherworld significantly reduces the risk of data loss due to an operating system

crash by preserving the in-memory data tables across kernel crashes. Because the database server communicates with other processes, Otherworld cannot currently resurrect it without a crash procedure. By adding a simple crash procedure that saves the contents of the in-memory database to the disk and restarts the database server, we can improve its fault tolerance without introducing runtime overhead or architectural changes.

For our tests, we used MySQL. The MySQL architecture isolates the code responsible for maintaining data at the physical level into a separate component called *pluggable storage engine (PSE)*, which is responsible for the low-level functions that store and retrieve data. MySQL supports different types of PSEs. One of these, called MEMORY PSE, stores the table data in memory without saving it to disk, thus making the database memory-resident in order to improve performance.

By examining the MEMORY PSE source code, we found that all tables allocated by MEMORY PSE are organized internally in a linked list, which is accessible through a global variable. MEMORY PSE has functions used for scanning and retrieving row data from the tables, returned in an internal format. Also, it has a function that accepts data in the same format as an argument and inserts it as a new row in the table. For the purposes of writing our crash procedure, we did not need to know how these functions work or the format of row data.

The crash procedure for the MySQL server iterates through the list of all allocated tables, calls the appropriate functions to retrieve data rows from these tables, and saves them to disk. Since the row format is not relevant for our purposes, we interpret the row contents as an array of bytes. After the crash procedure has saved all data to disk, it restarts MySQL, passing it the name of the file with the saved data in the command line. Furthermore, we modified MySQL to *(i)* read during startup the content of all MEMORY PSE tables from disk that were saved earlier by the crash procedure, and *(ii)* initialize the in-memory tables with this content.

It took us a day to write this simple crash procedure. Most of this time was spent on getting ourselves familiar with the MySQL architecture, since we did not have any prior experience with this product. Overall, MySQL has about 700,000 lines of code, and our modifications consisted of 70 new and 5 modified lines of code.

5.3 Web Application Servers

While the HTTP protocol is stateless, many web applications need a way to maintain session data, such as the contents of a shopping cart or user credentials, across a sequence of page accesses. Some Web applications need to be fault tolerant, which means that user session data cannot be lost on system failures. To address this requirement, session data is typically stored on disk or in a database. The copying of session data between in-memory representation and persistent storage causes at least a 25% performance decrease [23]. By adding an Otherworld crash procedure to the Web application server, we can prevent losing session data on kernel failures without the overhead of going to persistent storage. Once a crash procedure is added to the Web application server, no changes should be required to any Web application that runs on this server.

For our case study, we selected the Apache and PHP bundle. The session data is stored by the PHP code in shared memory and is available to applications through PHP functions. PHP session code stores session data into a hash table using the session id as a key and a serialized version of the session data as a value. The address of the hash table is stored in a global variable. The crash procedure that we wrote gets the address of the hash table and saves each element of the table to a file. After the session data is saved to disk, Apache restarts and initializes the session data table from the file.

As in the MySQL case, we did not need to know how session data is serialized or the details of the session hash table implementation because we reused functions that already existed to retrieve and populate the session hash table. As a result, changes to the PHP code were limited to 110 new and 5 modified lines of code. All modifications were limited only to the PHP module code itself, so all PHP applications can benefit from improved fault tolerance without any changes.

5.4 In-memory Checkpointing

A popular mechanism for minimizing the consequences of application and operating system failures is checkpointing. There are several approaches aimed at reducing the overhead of checkpointing by saving checkpoints to memory rather than to a disk. Zheng et al. show that saving checkpoints to memory reduces overhead by more than a factor of ten [34]. However, in-memory checkpointing does not protect from operating system crashes because the memory is overwritten during a traditional system reboot. On the other hand, the advantage of checkpointing is that it does not necessarily require support from the applications. By combining Otherworld with existing checkpointing techniques, we can improve the reliability of in-memory checkpointing by protecting in-memory checkpoints from operating system crashes without changing the applications themselves.

We tested our technique with the Berkeley Labs Checkpoint-Restart (*BLCR*) library [16]. BLCR is a system-level checkpointing solution. BLCR consists of a kernel module and a user-level library that together checkpoint unmodified applications. We modified BLCR so that, instead of writing checkpoints to disk, it writes them to memory. We measured the performance of the in-memory checkpointing solution against the performance of unmodified BLCR. As long as the checkpoint size is significantly less than the amount of available physical memory, checkpointing performance improves approximately by a factor 10. We were also able to successfully recover application checkpoints from operating system crashes and continue running applications from those checkpoints. We did not introduce any modifications apart

from modifying BLCR to keep checkpoints in memory. That is, no crash procedure was necessary in this case.

6. Evaluation

In order to evaluate the reliability of Otherworld, we utilized the synthetic fault injection mechanism originally developed at the University of Michigan for evaluating the reliability of the Rio File Cache [25] and later used for evaluating Nooks reliability [30]. Each fault changes a single integer value on the kernel stack of a random thread, or a single instruction, or instruction operand in the kernel code. This emulates many common errors, such as stack corruption, uninitialized variables, incorrect testing conditions, incorrect function parameters, and wild writes.

In order to simplify the automation of a large number of test runs, we conducted experiments that did not measure performance within a VMWare virtual machine. The machine had two virtual CPUs, 1GB of RAM, and 22GB of disk storage. Experiments that measured performance were run on a physical machine with a single dual core CPU, 4GB of RAM and 120GB of disk storage.

We tested the reliability of Otherworld with 5 applications: the vi and JOE text editors, MySQL, Apache/PHP, and the BLCR in-memory checkpointing system. For each application, we conducted a number of fault injection experiments. For each experiment we injected 30 faults at a time. About 20% of the experiments did not result in a kernel fault, and all applications continued executing with no visible problems. We discarded these experiments from our statistics. In total we observed 400 experiments for every application that ended with a kernel fault. In each experiment, we waited until the main kernel initialized itself, loaded the crash kernel, and started the application before injecting the faults. Each application executed a workload, and the progress of this workload was logged on a remote computer so that we could know the correct state of the application at every point in time. We injected faults after a random amount of time and observed the outcome. If the system was unable to perform a microreboot, or the crash kernel failed to resurrect the application because of main kernel memory corruption, we considered resurrection a failure. After an application was resurrected, it saved its data to disk, and we checked this data against the remote log. If any data was missing or incorrect, we considered resurrection to have failed because of data corruption. Otherwise, we considered resurrection to be a success. Since no extra code is executed unless a crash occurs, we did not observe any run-time processor or I/O overhead.

The vi and JOE workloads consisted of replaying a sequence of keystrokes that emulated a working user. The MySQL workload consisted of a sequence of SQL queries issued by a remote client that inserted, deleted, and updated data in the in-memory table. After resurrection, we checked the correctness of data within the table. The BLCR workload consisted of periodic in-memory checkpointing of a test application with a memory footprint of 800MB. After resurrection, we ensured that the application could be restored from the checkpoint and that application data was not corrupted. We were able to successfully resurrect applications in more than 97% of the cases .

Benchmark	Increase in TLB misses	Performance overhead
MySQL	22%	3.4%
Apache	51%	4.8%
Volano	55%	11.6%

Table 3. Performance overhead of enabling user memory space protection while executing system calls.

Application	Kernel memory	Page tables
vi	116 KB	60%
JOE	137 KB	61%
MySQL	711 KB	70%
Apache	844 KB	83%
BLCR	941 KB	83%

Table 4. Size of the data read by the crash kernel during the resurrection process.

Our first set of tests showed a successful resurrection rate of only 89%. In order to improve this number, we did several incremental modifications that improved the robustness of Otherworld. We found that in 8% of the experiments, the system either completely stalled or experienced recursive failures, thus failing to transfer control to the crash kernel. Also, we found that in response to double faults (an exception that occurs if the processor encounters a problem while trying to service a pending interrupt or exception) KDump stops the system instead of invoking the crash kernel. We fixed the first problem by enabling Linux software lock detection and emulating a hardware watchdog timer by forcing the virtual machine to issue a non-maskable interrupt (*NMI*) on detection of a stall. The NMI handler responds with microrebooting the kernel. Although not widely used in practice, the hardware watchdog timer is a common component of many modern x86 and ARM chipsets. Intel has included a watchdog timer in all of its x86 architecture chipsets since 2002 [18]. AMD also includes a watchdog timer in many of its chipsets. Both Linux and Windows are shipped with watchdog drivers. The second problem was corrected by fixing the Linux double fault interrupt handler to start the microreboot process. Other fixes to KDump included prevention of infinite recursion while trying to print the stack and not relying on the validity of the descriptor of the currently executing process. After these changes, the successful resurrection rate increased to 97% or better. None of those changes affected application performance.

Application	Successful resurrection	Failure to boot the crash kernel	Failure to resurrect application	Data corruption with / without user space protected
vi	97.5%	2.5%	0%	0% / 0%
JOE	97.75%	2.25%	0%	0% / 0.25%
MySQL	97.25%	2%	0.5%	0.25% / 0.5%
Apache/PHP	97%	3%	0%	0% / 0%
BLCR	97%	2.75%	0.25%	0% / 0.5%

Table 5. Results of resurrection experiments.

We did not encounter any application data corruption for Apache/PHP, but we encountered data corruption on one occasion for JOE and on two occasions for BLCR and MySQL. Protecting the user space, as described in Section 4, reduced (for MySQL) or even eliminated (for the rest of workloads) user space corruption in our experiments but introduced overhead mainly due to TLB flush operations that occur on every page table switch. In order to estimate the performance impact of protecting user memory spaces while executing code in the kernel, we ran the MySQL benchmark suite and the Apache benchmarking tool with and without user space protection enabled. Since neither in-memory checkpointing systems nor text editors have a high rate of system calls, they were not affected by page switching overhead and were not further considered for this evaluation. However, we added the Volano benchmark to our tests [33]. This benchmark simulates a chat server with multiple client sessions. It is a highly parallel and system call intensive application, the type of workload that should be the most sensitive to system call overhead. The results of these experiments are presented in Table 3. As we can see, the overhead of protecting the user space ranges from 3%-5% for Apache and MySQL to 11.6% for Volano benchmark.

We were able to detect only 3 cases, out of a total of 2000 experiments, where resurrection failed due to kernel data structure corruption. This result is perhaps to be expected, since the amount of data needed for resurrection from the main kernel is relatively small. Table 4 shows the size of the data that the crash kernel needs to read from the main kernel for resurrecting the applications we tested. We found this size to be less then 1 MB for all of the examples considered. The last column lists which proportion of the main kernel data structures required to resurrect the process contained page tables, illustrating that page tables constitute the largest portion of the main kernel data retrieved. The ratio between the size of the main kernel data important for resurrection and the size of the virtual address space gives us a rough estimate of the probability of wild writes corrupting data important for resurrection. Even for an application with the largest possible memory footprint on a 32-bit systems - 3 GB, the amount of data retrieved will be approximately 5 MB, which is less than 0.13% of the total address space.

Application	Boot time	Service interruption time
shell	64	53
MySQL	71	64
Apache	70	68

Table 6. Service interruption time (seconds).

The results are summarized in Table 5. The second column contains the percentage of cases in which Otherworld successfully preserved application data through the resurrection after a failure. The third column lists the percentage of cases where Otherworld failed to boot the crash kernel. The fourth column lists the percentage of cases where corruption in the main kernel structures was detected, preventing resurrection. The last column lists the percentage of cases where applications were successfully resurrected, but subsequent data verification detected data corruption. This column contains two numbers. The first represents the percentage of cases where application data was corrupted while running with application memory protection, while the second is without application memory protection. For each application, Otherworld was able to recover application data successfully in 97% or more of the cases.

The major source of resurrection failure is the inability to transfer control from the main to the crash kernel. Although the amount of code involved is minimal, Otherworld still requires coordination between CPUs on multiprocessor systems and is sensitive to the corruption of certain kernel page entries and the interrupt descriptor table. Since the crash kernel is kept uninitialized and is protected by the memory hardware, we found that once we succeed in passing control to the crash kernel, it successfully boots itself in 100% of the cases.

When user space was not protected, 5 experiments out of 2000 (less then 0.2%) ended with application data corruption. Protection of user space, as described above, introduces overhead but significantly reduces the probability of undetected corruption. With protection enabled, application corruption was observed only in one MySQL experiment, due to a undetected corruption of a page table entry.

Finally, we measured the time for the system to recover from a failure while running different workloads. The results are presented in the Table 6. The second column contains

the time of a system cold start, from the time of pressing the power button to the time the workload is operational. The third column contains the time from when the workload is interrupted by a failure to the time the workload is resurrected and operational again. The first row contains the time till the interactive user is presented with the text mode shell without any additional application start or resurrection. The second and the third rows show the time till MySQL and Apache are operational. Since the crash kernel initializes itself after the failure from scratch, the time it takes to boot the crash kernel is comparable to the time of a cold system start excluding the time consumed by the BIOS and boot loader initialization. Both Apache and MySql resurrection involves calling the crash procedure to save the application data and the restart of the application. Because of this, the resurrection process is longer than a clean application start. This makes the time during which the workload is not operational comparable with that of a full system reboot. Currently, we a considering different methods of reducing service interruption time after a kernel failure. For example, the exact hardware configuration information is known by the time of a crash, so the crash kernel hardware detection procedure may be significantly simplified.

7. Conclusions and Future Work

In this paper, we have introduced Otherworld, a mechanism that on an operating system kernel failure: *(i)* microreboots the operating system without clobbering the state of the applications, *(ii)* restores the application processes along with their memory, open and mapped files, signal handler descriptors, and physical terminals, and *(iii)* continues the execution of these processes from the point at which they were interrupted, if restoration was successful. Otherworld thus significantly increases the level of fault tolerance. In the vast majority of cases, the resurrected applications can at minimum preserve their data to disk and restart, if they cannot continue their execution across the kernel failure. This is in stark contrast to the current state of affairs, where a kernel failure results in a full system reboot with the loss of all volatile application state.

We implemented Otherworld in Linux with only minor changes to the kernel and existing applications. We tested Otherworld using a variety of different application types and showed that, even with some kernel resources used by applications not being restored, all of the above applications were able to restore their data in more than 97% of kernel faults. Either no changes to the applications were required or the changes were minimal and straightforward.

The key benefits of our technique include zero overhead (or small runtime overhead when application memory space is protected) and small and fixed memory overhead that is independent of the amount of data used by the applications. Another key element of Otherworld is that it does not depend on a specific operating system architecture. It can be used with existing commercial operating systems with monolithic kernels, such as Windows or BSD Unix, as well as with microkernel operating systems. This fact is crucial for an industry where billions of dollars have been invested in legacy operating systems.

Making it feasible for applications to tolerate kernel faults with no overhead offers new ways to significantly improve performance of some applications, such as databases or checkpointing libraries, by allowing them to keep all their data in memory with significantly reduced risk of losing the data due to an operating system fault.

As future work, we intend to add support to make additional resource types resurrectable. In particular, we are currently working on the automatic resurrection of TCP/UDP/IP sockets and pipes. We also intend to make various performance optimization to Otherworld's current implementation. Examples include reducing the initialization time of the crash kernel (for example, by executing some of the initialization code immediately after the crash kernel is installed and by exploiting device information from the crashed main kernel), and using page mapping techniques to reduce the amount of data copied. Finally, we are investigating ways to efficiently detect and prevent kernel and application data corruption that might be caused by a kernel fault.

We expect that the reliability of operating systems may well improve in the future. Even then, we believe that Otherworld will still be useful, for example, by allowing the kernel to microreboot without terminating running applications, Otherworld may also be used for hot updates of an operating system running mission critical software that cannot afford restarts. Provided that service interruption time during the kernel reboot can be improved, this feature can be also used for fast system rejuvenation.

8. Acknowledgments

We would like to thank our shepherd, Julia Lawall, and the anonymous reviewers for their excellent feedback and helpful suggestion for how to improve this paper. We thank our colleagues Reza Azimi, Adam Czajkowski, Livio Soares, and David Tam for their help and insightful comments.

References

[1] Auslander, M., Larkin, D., and Scherr, A. The evolution of the MVS operating system. *IBM Journal of Research and Development 25, 5* (1981), 471–482.

[2] Avizienis, A., Laprie, J. C., Randell, B., and Landwehr, C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing 1, 1* (2004), 11–33.

[3] Baker, M., Asami, S., Deprit, E., Ouseterhout, J., and Seltzer, M. Non-volatile memory for fast, reliable file systems. *Proc. of the 5th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (1992), 10–22.

[4] Baker, M., and Sullivan, M. The Recovery Box: Using fast recovery to provide high availability in the Unix environment. *Proc. of the 1992 USENIX Summer Conf.* (1992), 31–43.

[5] Baumann, R. Soft errors in commercial semiconductor technology: overview and scaling trends. *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals* (2002), 121.

[6] Biederman, E. Kexec. http://lwn.net/Articles/15468/, 2002.

[7] Bohra, A., Neamtiu, I., Gallard, P., Sultan, F., and Iftode, L. Remote repair of operating system state using Backdoors. *Proc. of the Intl. Conf. on Autonomic Computing* (2004), 256–263.

[8] Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., and Fox, A. Microreboot–a technique for cheap recovery. *Proc. of the 6th Symposium on Operating Systems Design and Implementation* (2004), pp. 31–44.

[9] Chandra, S., and Chen, P. M. The impact of recovery mechanisms on the likelihood of saving corrupted state. *Proc. of the 13th Intl. Symposium on Software Reliability Engineering* (2002), 91–101.

[10] Chen, Y., Gnawali, O., Kazandjieva, M., Levis, P., and Regehr, J. Surviving sensor network software faults. *Proc. of the 22nd Proc. of the Symposium on Operating Systems Principles* (2009), 235–246.

[11] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. An empirical study of operating system errors. *Symposium on Operating Systems Principles* (2001), 73–88.

[12] David, F. M., Chan, E. M., Carlyle, J. C., and Campbell, R. H. CuriOS: Improving reliability through operating system structure. *Proc. of the 8th Symposium on Operating Systems Design and Implementation* (2008), 59–72.

[13] Depoutovitch, A., and Stumm, M. Otherworld - giving applications a chance to survive OS kernel crashes. *Proc. of the 4th Workshop on Hot Topics in System Dependability* (2008).

[14] Goyal, V., Biederman, E., and Nellitheertha, H. KDump, a Kexec-based kernel crash dumping mechanism. *Proc. of the Linux Symposium* (2005), 169–181.

[15] Gu, W., Kalbarczyk, Z., Iyer, R., and Yang, Z. Characterization of Linux kernel behavior under errors. *Proc. of the Intl. Conf. on Dependable Systems and Networks* (2003), 459–468.

[16] Hargrove, P., and Duell, J. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *Journal of Physics: Conf. Series* (2006), vol. 46, Institute of Physics Publishing, pp. 494–499.

[17] Herder, J. N., Bos, H., Gras, B., Homburg, P., and Tanenbaum, A. S. Reorganizing Unix for reliability. *Proc. of Asia-Pacific Computer Systems Architecture Conf.* (2006), 81—94.

[18] Intel. Using the Intel ICH family watchdog timer (WDT) application note: AP-725 *http://www.intel.com/design/chipsets/applnots/292273.htm* (2002)

[19] King, S., Dunlap, G., and Chen, P. Debugging operating systems with time-traveling virtual machines. *Proc. of the USENIX 2005 Technical Conf.* (2005), 1–15.

[20] Laadan, O., and Nieh, J. Transparent checkpoint-restart of multiple processes on commodity operating systems. *Proc. of the 2007 USENIX Technical Conf.* (2007), 323–336.

[21] Lehman, T., Shekita, E., and Cabrera, L. An evaluation of the Starburst memory-resident storage component. *IEEE Trans. on Knowledge and Data Engineering* (1992), 555–566.

[22] Lowell, D. E., Chandra, S., and Chen, P. M. Exploring failure transparency and the limits of generic recovery. *Proc. of the 4th Symposium on Operating System Design and Implementation* (2000), 289–304.

[23] Microsoft. Underpinnings of the session state implementation in ASP.NET. http://msdn2.microsoft.com/en-us/library/aa479041.aspx, 2003.

[24] Ng, W. Design and implementation of reliable main memory. *Ph.D. thesis* (1999).

[25] Ng, W. T., and Chen, P. M. The systematic improvement of fault tolerance in the Rio file cache. *Proc. of the 1999 Symposium on Fault-Tolerant Computing* (1999), 76–83.

[26] Patterson, D. Recovery oriented computing: A new research agenda for a new century. *Proc. of the 8th Intl. Symposium on High-Performance Computer Architecture* (2002), 223.

[27] Srinivasan, S., Andrews, C., Kandula, S., and Zhou, Y. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. *Proc. of the USENIX 2004 Annual Technical Conf.* (2004), 29–44

[28] Sullivan, M., and Chillarege, R. Software defects and their impact on system availability: A study of field failures in operating systems. *Proc. of the 21st Intl. Symposium on Fault-Tolerant Computing* (1991), 2–9.

[29] Swift, M. M., Annamalai, M., Bershad, B. N., and Levy, H. M. Recovering device drivers. *ACM Transactions on Computer Systems 24, 4* (2006), 333–360.

[30] Swift, M. M., Bershad, B. N., and Levy, H. M. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems 23, 1* (2005), 77–110.

[31] Van Vleck, T. Unix and Multics. http://www.multicians.org/unix.html, 1993.

[32] VMWare Fault tolerance, http://www.vmware.com/

[33] Volano benchmark, http://www.volano.com/benchmarks.html

[34] Zheng, G., Shi, L., and Kalé, L. FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. *Proc. of the 2004 IEEE Intl. Conf. on Cluster Computing* (2004), 93–103.