Mini-Ckpts: Surviving OS Failures in Persistent Memory

David Fiala, Frank Mueller
North Carolina State University*
davidfiala@gmail.com,
mueller@cs.ncsu.edu

Kurt Ferreira
Sandia National Laboratories†
kbferre@sandia.gov

Christian Engelmann
Oak Ridge National Laboratory‡§
engelmannc@ornl.gov

ABSTRACT

Concern is growing in the high-performance computing (HPC) community on the reliability of future extreme-scale systems. Current efforts have focused on application fault-tolerance rather than the operating system (OS), despite the fact that recent studies have suggested that failures in OS memory may be more likely. The OS is critical to a system's correct and efficient operation of the node and processes it governs — and the parallel nature of HPC applications means any single node failure generally forces *all* processes of this application to terminate due to tight communication in HPC. Therefore, the OS itself must be capable of tolerating failures in a robust system. In this work, we introduce `mini-ckpts`, a framework which enables application survival despite the occurrence of a fatal OS failure or crash. `mini-ckpts` achieves this tolerance by ensuring that the critical data describing a process is preserved in persistent memory prior to the failure. Following the failure, the OS is rejuvenated via a warm reboot and the application continues execution effectively making the failure and restart transparent. The `mini-ckpts` rejuvenation and recovery process is measured to take between three to six seconds and has a failure-free overhead of between 3-5% for a number of key HPC workloads. In contrast to current fault-tolerance methods, this work ensures that the operating and runtime systems can continue in the presence of faults. This is a much finer-grained and dynamic method of fault-tolerance than the current coarse-grained application-centric methods. Handling faults at this level has the potential to greatly reduce overheads and enables mitigation of additional faults.

*This work was supported in part by a subcontract from Sandia National Laboratories and NSF grants CNS-1058779, CNS-0958311.

†Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

‡This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

§This work was sponsored by the U.S. Department of Energy's Office of Advanced Scientific Computing Research.

ICS '16, June 01-03, 2016, Istanbul, Turkey
© 2016 ACM. ISBN 978-1-4503-4361-9/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2925426.2926295

1. INTRODUCTION

The operating system (OS) kernel is responsible for supervising running applications, providing services, and governing hardware device access. While applications are isolated from each other and a failure in one application does not necessarily affect others, a failure in the kernel may result in a full system crash, stopping all applications without warning due to system services that run across the entire machine

In current high-performance computing (HPC) systems with 100s of thousands of processor cores and 100s of TB of main memory, faults are becoming increasingly common and are predicted to become even more common in future systems. Faults can affect memory, processors, power supplies, and other hardware. Research indicates that servers tend to crash twice per year (2-4% failure rate) and unrecoverable DRAM errors occur in 2% of all DIMMs per year [37]. A number of studies indicate that ECC mechanisms alone are unlikely to be sufficient in correcting a significant number of these memory errors [26, 38]. While not all faults result in

failure, when they do, intermediate results from entire HPC application executions may be lost [1].

The objective of this work is to isolate applications from faults that originate within the processor or main system memory and result in OS kernel failures and crashes. This work is critical to the scalability of current and future large-scale systems due to the fact that while the kernel occupies a relatively small portion of memory (10-100 MB) relative to an HPC application (up to 32 GB per node), errors in kernel memory may be more likely [26], and the kernel is critical to all other processes running on a node — and in HPC even other nodes whose progress depends on being able to communicate with the faulted node. In this work, we target the Linux OS due to its widespread adoption on current HPC platforms (97% of the Top500 [5]), although the methodology described in this work applies directly to most other OSs.

While failures due to hardware faults are a concern in large clusters, software bugs pose an equally fatal outcome in the kernel, regardless of scale. Bugs may result in kernel crashes (a kernel "panic"), CPU hangs, and memory leaks resulting in eventual kernel crashes, or other outcomes leaving system performance degraded or only partially functional. Therefore, we have designed `mini-ckpts` to recover the system in the face of both kernel bugs and transient hardware faults.

The primary objective of `mini-ckpts` is to demonstrate the feasibility of voluntary or failure-induced system reboots by serializing and isolating the most essential parts of the application to persistent memory. To this end, we make minimal modifications to the OS for saving the CPU state during transitions between user space and kernel space while migrating application virtual memory to an execute-in-place (XIP) directly mapped physical memory region. Finally, failures are mitigated by warm booting an identical copy of the kernel and resuming the protected applications via restoring the saved state of user processes inside the newly booted kernel.

While other research, such as Otherworld [14], has proposed the idea of a secondary crash kernel (activated upon failure) that parses the data structures of the original, failed kernel, this work is novel as it is the first to demonstrate complete recovery independence from the data structures of an earlier kernel. The premise is that a kernel failure may corrupt the kernel's content. The failure itself might be due to a corruption or bug in the first place, which could impede access to the old kernel's data. `mini-ckpts`' unique contribution is that it has no dependencies on the failed kernel, and that all state required to transparently resume a protected process can be proactively managed prior to a failure. This allows HPC applications to avoid costly rollbacks (during kernel panics) to previous checkpoints, especially when considering that rolling back on a massively-parallel HPC system typically requires all communicating nodes to coordinate and rollback together unless coupled with other more costly techniques such as message logging [30].

This work makes the following contributions:

- It identifies the minimal data structures required to save a process' state during a kernel failure.
- It further identifies instrumentation locations generically needed in OSs for processor state saving.
- It introduces `mini-ckpts`, a framework that implements process protection against kernel failures by saving application state in persistent memory.
- It evaluates the overhead and time savings of `mini-ckpts`' warm reboots and application protection.
- It experimentally evaluates application survival of `mini-ckpts` protected programs in the face of injected kernel faults and bugs for a number of key OpenMP and MPI parallelized workloads.

2. MINI-CKPTS OVERVIEW

`mini-ckpts` protects applications by taking a checkpoint of the process and its resources, modifying its address space for persistence, and then periodically saving the state of the process' registers whenever the kernel is entered. With the checkpointed information and register state, `mini-ckpts` later recovers a process after a kernel crash occurs. Recovery is performed by attempting to recreate the process in the exact state prior to the crash using well established techniques employed by traditional checkpoint/restart software, such as Berkeley Lab Checkpoint/Restart (BLCR) [15]. The basic checkpointing features `mini-ckpts` supports are shown in Table 1. Additionally, `mini-ckpts` support for persistent virtual memory is comprehensive. A summary is available in Table 2.

To use `mini-ckpts`, one first needs to boot a kernel with `mini-ckpts` modifications followed by additional bootstrapping steps during start-up:

Table 1: Summary of process features `mini-ckpts` protects in checkpoint during failures

Multi Threaded Proc.	Process ID
Mutex/Conditions Variables	Process UID & GID
Regular Files (excl. seek)	System Calls
Signal Handlers & Masks	Unflushed File Buffers
Stdin/Out/Err	mmap'd files
Block Dev. /dev/{null,zero}	mprotect
FPU State	CPU Registers
Process Credentials	Process Parent/Child

Table 2: Persistent virtual memory support protected during failures

BSS, Data, & Heap sections	Executable (text)
Anonymous mmap'd Regions	Shared Libraries
File-based mmap Regions	vdso, vsyscall

[1]While a failure is one possible consequence of a fault, we do not strictly distinguish these terms in the following since the common term "kernel fault" actually refers to a kernel failure resulting from a fault, i.e., the terminology is used inconsistently in practice but the context allows one to infer the meaning.

- The `mini-ckpts` kernel is booted and filesystems such as */proc* and */dev* are mounted.
- The PRAMFS persistent memory filesystem is either initialized as empty or mounted if it already exists from a prior boot.
- A fresh copy of the `mini-ckpts` kernel is loaded into a reserved section of memory for later use (upon a system crash).
- If we are booting after a failure, the last checkpoint is restarted automatically. The `mini-ckpts` protected application resumes exactly where it left off.

Once an application is launched on a `mini-ckpts` enabled kernel, failure protection begins after the first checkpoint operation. A checkpoint operation is either self-triggered programmatically via a call to a shared library (preloaded with `LD_PRELOAD` or triggered by sending a specific signal to the process. The initial checkpoint migrates the process' virtual memory to the protected and persistent file system (PRAMFS), described in the following section. It then stores its open files, memory mappings, signal handlers, and other state detailed in Tables 1 and 2. Once the first checkpoint operation is complete, the system will ensure that the CPU registers and FPU (Floating Point Unit) state are always saved in persistent memory when the kernel is entered. Any further kernel entries, whether system call or interrupt handler, will further update the saved checkpoint with the correct CPU and FPU state. Once a checkpoint is committed to the persistent stable storage, any fault in the kernel will automatically warm reboot the system and trigger the application to resume exactly at the previous kernel entry which failed. This process can repeat indefinitely until the `mini-ckpts` protected application completes its execution successfully.

3. PERSISTENT VIRTUAL MEMORY ACROSS REBOOTS

An application's virtual memory consists of file-mapped memory regions, anonymous ("private" or lacking a file backing) memory regions, and special (vsyscall or VDSO) regions. Also, file-mapped memory regions may be either private CoW (copy on write) or shared between processes. Some memory regions are subject to write modifications during program execution (e.g., "anonymous" regions: data, stack, heap space).

Internally, the kernel provides applications with anonymous memory virtual pages by designating frames from unused physical memory. The physical to virtual layout and structure is stored within the kernel and is volatile, i.e., can be lost or corrupted should the kernel be rebooted. Another volatile memory type is `tmpfs`, the 100% in-memory RAM file system. Although `tmpfs` is entirely stored in physical memory, it is incapable of surviving a kernel restart because the mapping that describes both the layout and contents of files are not stored in persistent memory.

3.1 PRAMFS: Protected and Persistent RAM Filesystem

Our approach requires a 100% in-memory RAM filesystem that can survive a kernel crash and reboot to persist anonymous and/or private memory regions across kernel failures without the need for perfect kernel data structure state.

PRAMFS [4] is a persistent NVRAM filesystem for Linux that is capable of storing both the filesystem's metadata as well as contents 100% in-memory while bypassing the Linux Page Cache. A PRAMFS partition can be utilized on volatile RAM as well as NVRAM, although the lifetime of a PRAMFS partition would then be restricted to the duration of power being applied to the RAM (i.e., a PRAMFS filesystem in RAM cannot survive once the power is turned off). PRAMFS can be combined with file mapping in processes as well as execute-in-place (XIP) support to allow for a process' file-backed mappings to not only read/write directly to physical memory but also execute directly from PRAMFS [25]. Since memory mappings to PRAMFS are one-to-one with physical memory, an application whose executable, data, stack, and heap are mapped to a file based in PRAMFS would, in fact, be entirely executing, reading, and writing within the PRAMFS storage without any volatility should the process be terminated at any point. To put this another way, if all of a process' memory was mapped to PRAMFS files and the process was terminated prematurely at any arbitrary point, then the image in PRAMFS would be very similar to a process' core dump.

3.1.1 Protecting PRAMFS from Spurious Writes During Faults

Some faults may result in spurious writes to parts of memory related to or referenced by the faulting code. `mini-ckpts` assumes that the memory stored in PRAMFS is guarded by user space protection mechanisms, such as algorithm-based fault tolerance, software fault tolerance, redundancy, or some other means outside the scope of OS resilience [23, 20]. Otherworld investigated the use of write-protected page tables to protect user space memory and reported a runtime overhead between 4%-12% for this additional protection [14] while protecting NVRAM may result in 2x to 4x runtime overheads [22].

3.2 Preparing a Process for Mini-Ckpts

`mini-ckpts`' goal is to ultimately preserve process state in the event of a kernel crash, therefore, it is imperative that the contents of all memory mappings are safely retained. However, there are several regions of memory that will be lost during a kernel crash, mostly due to their dependence on the volatile page-cache, such as copy-on-write mappings, private (anonymous) mappings, data, and the stack.

To always ensure consistency of non-volatile backing

store of applications, `mini-ckpts` migrates all memory mappings to PRAMFS so that the process' memory may survive a kernel crash. To achieve this, we added functionality to the kernel to pause all running threads of a process, iterate through each virtual memory mapping of the process, and either copy or move each memory mapping to a new, distinct file in PRAMFS. For each memory mapping we created a new file in PRAMFS sized to match each distinct memory region. Then, for each region we copy the contents of the existing memory to the file, unmap the old region, and then remap the new PRAMFS file into the same location with the same read, write, and/or execute flags, but with a shared instead of a private mapping, which ensures that writes instantly update the PRAMFS region. Note: Once the PRAMFS files are virtually mapped into a process, there are no further dependencies within the kernel to maintain the mappings. The hardware accesses the page tables directly, and even a failing kernel would not necessarily affect an application's direct access to its memory as there is no buffering/caching between the application, kernel, and physical memory.

4. INITIAL CHECKPOINTING

Protection for applications in `mini-ckpts` begins with the first initial checkpoint. This initial checkpoint performs two major functions; first, it remaps all memory into shared memory that is backed in PRAMFS and second, it stores a serialized version of the process' state. A checkpoint can be triggered either locally (synchronously) or through another process (via a signal).

The initial checkpoint modifies its memory mappings. To ensure that the application is not running during this time, `mini-ckpts` interrupts all threads of the target process with a signal and invokes a system call within the threads' signal handlers. The system call traps all threads in the kernel and puts them to sleep until the checkpoint commit operation is complete. Once within the kernel, `mini-ckpts` remaps all process memory to PRAMFS as previously described. All memory regions, except the VDSO and vsyscall, are individually mapped to separate files in PRAMFS. Since memory regions may be several GB in size, which exceeds PRAMFS' limits, `mini-ckpts` maps large contiguous mappings into separate files, each no larger than 512MB.

After remapping process memory, `mini-ckpts` records process state information to a separate persistent region of memory. This recording largely mirrors the BLCR checkpoint library's storing of information, such as open file descriptors, threads, virtual memory layout and protections, process IDs, signal handlers, etc. (see [15] and Table 1). TCP sockets are not restarted automatically, which we address later in the context of how to restore MPI communication.

Before allowing the application to resume, `mini-ckpts` marks each running thread as tracked within the kernel. This will enable `mini-ckpts` to capture the registers of this thread every time it is interrupted by the kernel. Thus, the kernel will always maintain a safe copy of each thread's registers prior to a possible kernel crash. This allows the threads to restart immediately without any loss in computation upon a kernel failure.

5. CAPTURE AND RESTORATION OF REGISTERS

A `mini-ckpts` checkpoint is taken whenever a thread relinquishes control over a processor: at a (1) system call, (2) interrupt, (3) non-maskable interrupt triggered by signals, scheduling, sleeping, synchronization (waiting for mutual exclusion) and both blocking and non-blocking system calls. Each type of transition makes a guarantee as to the state of the processor and registers when control is returned to the interrupted process even though some may change certain registers. When the hardware encounters an error, or when there is a software bug, such as a stuck processor, non-maskable interrupts may be triggered to initiate the `mini-ckpts` recovery. The only locations that require kernel-level instrumentation in order to save the registers of an interrupted process are the entry points to both, system calls and interrupt handlers, within the kernel.

5.1 Implementing Mini-Ckpts at the Interrupt and System Call Level

`mini-ckpts` needs to save the general purpose registers while entering the kernel. An interrupt already provides most of the functionality needed. `mini-ckpts` simply calls a routine that copies the saved registers to the `mini-ckpt` location in persistent memory. If a kernel crash occurs during the later handling of an interrupt, then the most recent register state is safely stored. `mini-ckpts` also saves the state of the floating point registers besides general purpose registers.

System calls are handled in a slightly different manner than interrupts, because some general purpose registers containing arguments to system calls are not saved. Since `mini-ckpts` depends on saving the state of all registers, the entry point for system calls required modifications to allocate kernel stack space for the complete set of registers as well as to both save and restore the registers during the system call entry and exit points.

If a system call ultimately results in a kernel crash, then `mini-ckpts` will later restore the process' registers, but it still needs to consider what to do about the initiated system call. We provide two solutions:

- Return to the instruction immediately after the `SYSCALL` instruction. This simulates the interruption of a system call by setting the register containing the system call return value to `EINTR`, which is a standard Linux return value indicating that the kernel was interrupted and that the user process should try to repeat the system call.
- Try to repeat the system call using the exact same arguments immediately after restoring the thread.

We recommend the former approach because: (1) best programming practices require developers to check and handle the return values from system calls, (2) libraries may already repeat the call for the programmer if they detect `EINTR`, and (3) if the system call is repeated automatically by `mini-ckpts`, then it risks potentially containing a handle to an object that no longer exists (such as a socket) after a restore.

5.2 Restoring Registers during Restart

For each thread of a process that is restored after a restart, its general purpose registers must be returned to their state immediately prior to the failure. A helper routine is used to recreate each thread using standard `clone` system calls, and then each newly created thread requests its registers to be restored through a system call via an `ioctl` to `mini-ckpts`. To ensure that registers are not clobbered upon returning from a `syscall`, a trampoline was added that uses a combination of the user space stack and injected code to simulate a transparent restart. This builds on common platforms capabilities: (1) the ability for a system call to return to a specific address, and (2) the potential for the kernel to modify the virtual memory of a process.

Our technique resembles `setjmp/longjmp` and depends on using the stack to feed registers to an injected trampoline code:

- A small region of executable helper code is injected into the process being restored.
- A new thread is created for each one lost during the failure.
- Thread registers are restored via a system call.
- The user space stack pointer is retrieved from the registers stored during the system call entry.
- The stack pointer is advanced as the values of all previously saved registers are written directly to the user space stack.
- The kernel's saved return instruction pointer is set to a region of injected code. The system call will now return to a new location.
- The injected code restores each register by popping it from the stack, including the saved flags. The final `retq` instruction returns execution back to the point immediately prior to the previous kernel failure.

Once the trampoline finishes restoring the thread's registers, the process is unable to discern that it was interrupted at all except for time measurements spanning between the last `mini-ckpt` during failure, restart, and time up until the *retq* completes in the trampoline.

6. HANDLING KERNEL FAULTS

A failure in the kernel (panic) should result in a software fault handler known as a kernel panic. Within the Linux kernel, the default action of the panic handler is to attempt a backtrace of the current stack, print the current registers, print an error message, if provided, and then halt the system. A kernel panic may be triggered by memory hardware faults, software bugs, or assertion failures within kernel code. Depending on the severity of the failure, it might not be possible for the kernel to perform its final debug print messages prior to halting. Additionally, in multicore (SMP) systems, the panic handler is tasked with attempting to halt all other processors.

6.1 Warm Rebooting

The Linux kernel provides `kdump`, a mechanism wherein the panic handler attempts to perform a warm reboot into a secondary crash kernel and simultaneously preserve a copy of its own memory for later observation and debugging from the crash kernel. Because the intent of a crash kernel is merely to provide the most basic services in order to inspect or save the memory of the previous kernel, the system is warm booted in an unusual configuration: SMP is unavailable, and the core on which the new kernel is running is typically the same core that previously experienced a kernel panic. For the purposes of HPC applications, any warm reboot would require a stable system with all SMP cores available. Once in this state, recovering would normally require a full reboot through the BIOS.

To provide a fully functional SMP system after a kernel panic, `mini-ckpts` implements a migrate-and-shutdown protocol that is used during a failure. First, since a failure may occur on any core, `mini-ckpts` installs a specialized NMI (Non-Maskable Interrupt) handler once a panic is detected. Next, since other cores running in parallel may not yet be aware of the failure, the failing core sends an NMI to all other cores. This NMI forces all cores to immediately jump to the NMI handler. As part of the NMI handler, any and all `mini-ckpts` protected processes save their registers. Once in the NMI handler, core 0 takes the lead (if it was not already the first core to panic) since it must be the first CPU to perform a warm reboot. Once core 0 has detected all other cores to have signaled that they are halted, core 0 begins unpacking and relocating a fresh copy of the kernel, sets up basic page tables, passes memory mapping information for the PRAMFS and persistent regions, and jumps to the entry point of the new kernel. It has essentially performed an emergency shutdown of all cores and then completed the same steps a traditional bootloader would do to start an operating system.

6.2 Requirements for a Warm Reboot

Minimizing the data structures required to perform a warm reboot is critical to `mini-ckpts`' success. The code paths needed for a successful warm reboot begin at a call to `panic`. The paths to reaching *panic* are very broad but may include dereferencing invalid pointers or assertion failures within the kernel. Once panic is reached, `mini-ckpts` ensures that no further memory allocation will be required. Its dependencies then involve functions that typically execute between 1-5 in-

structions, such as shutting down the local APIC, high performance timers, and then signaling NMI interrupts for all cores. For NMI interrupts to work, we must also assume that the code paths and interrupt vectors themselves remain valid.

Within the NMI handler, we must assume that the per-CPU interrupt stacks remain available. This also assumes that hardware registers were not corrupted as part of the failure. Once in the NMI handler, if the panic has interrupted a `mini-ckpts` protected task, `mini-ckpts` must be able to access a 64-bit pointer in the thread's `thread_info` struct. It is fine if any other thread or task related members have been corrupted. The single pointer both indicates (if non-zero) that the thread is protected and, in doing so, points directly to where the interrupted threads' registers should be saved in persistent memory. Locating the `thread_info` struct is fortunately trivial if the thread's registers are valid since it is calculated based on the kernel's stack pointer.

Finally, `mini-ckpts` assumes that the persistent region of memory where `mini-ckpts` stores a serialized copy of the protected process remains safe. The checkpoint itself is typically under 100KB, which is small relative to today's main memory size. Additionally, software correction codes could be applied to the persistent memory if desired.

7. SUPPORTING MPI APPLICATIONS

Traditional checkpoint/restart tools do not support automatic reconnection of network sockets for applications during restart. Instead, they tend to provide a callback procedure for applications that moves the burden (and programming logic required) to the developer, if they wish to support the checkpoint restart paradigm. But kernels typically buffer both file I/O and network traffic so that an application may falsely believe that it has successfully sent data that is still buffered and then lost on a kernel crash. As a result, MPI implementations that do support checkpoint/restart may require network traffic to quiesce and all buffers drained before a checkpoint is taken. Moreover, MPI implementations consider the loss of communication with a peer process as a critical error forcing an entire job to terminate without saving its computation.

`mini-ckpts` supports HPC workloads via run-through / forward progress fault tolerance for MPI applications through librlmpi. librlmpi specifically handles transient network failures, network buffer loss, and incomplete message transmission, which all may occur during a local or remote (peer) kernel failure. librlmpi is reliable, in that it supports handling lost messages either on the network wire or kernel buffer, and in that it can tolerate a network failure during any point of its execution or any of its system calls.

7.1 librlmpi Internals

Internally, during normal execution, librlmpi depends on `poll`, `writev`, and `readv`. It uses a threaded progress engine that monitors the return values of system calls. It can detect when `mini-ckpts` has transparently warm rebooted the system by watching for an `EINTR` system call error as discussed earlier. Upon detection of a warm reboot, librlmpi resets any message buffers in transit (pessimistically assuming they failed), and reestablishes connections with its peers. Likewise, a peer that has not undergone a failure is able to accept a new connection from a lost peer and resume communication where it left off. Any message transmissions that were cut off during a failure are restarted. Any data that had been buffered in kernel send or receive buffers will be resent, as librlmpi employs positive acknowledgments between peers.

librlmpi uses internal message IDs to uniquely identify and acknowledge transmissions between peers. After a failure, the peer node that failed (and subsequently warm rebooted) reestablishes connections with its peers. Both peers then send recovery details on the last messages they received even if they were not acknowledged earlier, i.e., a message is only resent if it was (1) not yet sent, or (2) lost in the kernel buffer or network. During normal execution and recovery, any message IDs that preceded the most recent acknowledgment are marked as complete (i.e., `MPI_Wait` or `MPI_Test` finish) and the space occupied by the message may be reused, per the normal MPI interface.

librlmpi employs the equivalent of a ready-send transmission protocol. Although ready-send semantics are not enforced at the application level, librlmpi will not transmit an outgoing message until its peer has sent a receive envelope indicating it is ready and has a buffer available to receive the message. If an MPI application attempts to send a message prior to a receive being posted, the transmission will be delayed until the receive is posted. From an application's point of view, we follow the MPI standard and support properly written MPI applications without any modifications.

Overall, librlmpi combined with `mini-ckpts` *transparently* allows an MPI application to be interrupted by a kernel panic and resume execution without loss of progress. It does not require any modification to an MPI application. If an MPI application wishes to begin its `mini-ckpts` protection automatically (instead of using an external signal), an API is provided, which allows the application to initiate protection.

7.2 MPI and Language Support

librlmpi supports basic MPI point-to-point communication, many collectives, and provides specialized routines to allow an MPI application to enable `mini-ckpts` protection and inject kernel faults for testing kernel panic recovery.

8. EXPERIMENTAL SETUP

`mini-ckpts` is evaluated in both physical and virtu-

Benchmark	CG	EP	IS	LU	BT	FT	MG	SP	UA
OpenMP Input	B	B	C	A	A	B	B	A	A

Bench./App.	CG	EP	IS	LU	PENNANT	Clover Leaf
MPI Input	C	C	C	B	sedov (400 iter.)	clover_bm2_short.in

Table 3: Used benchmark classes (input/problem sizes) for NPB for 8 Threads / 4 MPI Tasks

(measured in seconds)	BIOS boot time	**Kernel boot total**	Netw driver+ NFS-root mount	Kernel misc	Software stack total	Cold total w/ BIOS	**Warm boot tot.**
AMD Bare Metal	37.4	**5.3**	1.5	4.8	0.7	50.3	**6.0**
Intel Bare Metal	50.8	**6.7**	3.0	3.7	0.7	73.0	**7.4**
AMD VM	—	**0.8**	< 0.2	< 0.6	3.0	—	**3.8**
Intel VM	—	**0.7**	< 0.2	< 0.5	1.3	—	**1.9**

Table 4: Observed Times for Both Cold and Warm Booting a `mini-ckpts`-enabled System

alized environments to demonstrate `mini-ckpts` effectiveness and application performance overheads. Fault injections are provided as callable triggers that deliberately corrupt kernel data structures or directly invoke a kernel panic. `mini-ckpts` is evaluated on both OpenMP parallelized applications and MPI-based applications using a prototype MPI implementation with librlmpi to handle network connection loss at any point during execution.

Our test environments used for evaluation include: (1) 4 nodes with AMD Opteron 6128 CPUs, (2) 1 node with an Intel Xeon E5-2650 CPU, and (3) KVM virtualized environments running on the same AMD and Intel hosts. (1) and (2) are referred to as bare metal in the results section. The bare metal environments are all booted in a diskless NFS root environment. The virtual machine environments use a "share9p" root filesystem with their hosts, which, for practical purposes, is similar to a diskless NFS root for the virtual machine. In all configurations we provide 128MB of memory for storing persistent checkpoint information (although typically <100KB is used) and an additional 4GB of memory for the PRAMFS partition, which is where the memory of protected processes will reside during execution. All systems have been configured conservatively to optimize boot time: the system's boot order prioritizes booting directly to our custom kernel, which bypasses network controller initialization and any potential scanning for other boot options.

The experimental operating system modified for `mini-ckpts` is based on Linux 3.12.33, and our modified user space checkpoint library is a branch of BLCR [15] version 0.8.6_b4. Our bare metal and virtual machine experiments both use the same root environment based on Fedora 22. The kernel is configured for loadable module support, but all drivers, etc. are compiled directly into the kernel to tune it for a cluster environment. The only modules that are loaded at runtime are our modified BLCR checkpointing module and modules that are used to inject faults during our evaluation. As a result, when evaluating boot times and methods, we omitted the use of an initial ramdisk. Our hypervisor is qemu-kvm 1.6.2.

The NAS Parallel Benchmarks (NPB) OpenMP and MPI Versions [8] are used to evaluate the effectiveness of `mini-ckpts` in multithreaded/MPI environments. NPB includes computational kernels whose performance is indicative of other large-scale workloads and is a common benchmarks suite used in HPC. NPB DC (Data Cube) is excluded as it depends on heavy disk I/O. The benchmark classes (input/problem sizes) used are listed in Table 3.

For MPI, we evaluate the performance of the CG (Conjugate Gradient), IS (Integer Sort), LU (Lower-Upper Gauss-Seidel solver), and EP (Embarrassingly Parallel) benchmarks, as others depend on MPI functionality not implemented within our librlmpi prototype, such as `MPI_Comm_split`. We further evaluate for MPI the PENNANT mini-app version 0.7, an unstructured physics mesh mini-application from Los Alamos National Laboratory [3], and the clover leaf mini-app version 1.0 from Sandia National Laboratories, an Euler equation solver [1]. The OSU Micro Benchmarks version 4.4.1 is used to evaluate librlmpi's point-to-point latency, bandwidth, and collective operations [2]. Open-MPI version 1.8.5 is included in the experiments to compare the performance of a mainstream MPI implementation against our prototype implementation, librlmpi, and to establish any relative difference. All MPI benchmarks are run across four AMD Opteron 6128 nodes using Gigabit Ethernet for communication.

9. RESULTS

We first measure the time required for a full vs. warm reboot on both a bare metal and virtualized systems (Table 4). The cost of booting measured from the BIOS to the stage that the bootloader is reached is approximately 37 and 50 seconds on our bare metal test systems. As explained in the previous experimental setup section, the systems' boot order was optimized by directly booting to our kernel. Once the bootloader is reached and the kernel begins executing, the boot process requires between 5-7 seconds before the kernel is fully initialized. Of this, device initialization accounts for the majority of time. The network card drivers require between 1-3 seconds to initialize (even though IP addressing is statically configured vs. DHCP and the addressing is configured by the kernel instead of user space utilities). The remaining time is spread out among the initializations of various kernel subsystems

and other devices that are configured relatively quickly. In virtualized environments on the same systems as the AMD and Intel bare metal measurements, we observe that the same kernel boots in less than one second. The primary difference is the lack of slowly initializing hardware devices: The Ethernet driver required less than 0.1 seconds to complete, and the disk driver (share9p) completed initialization and mounting in under 0.2 seconds. `mini-ckpts` never requires a full reboot in either a virtualized or bare metal environment regardless whether or not it is activated to simply rejuvenate a kernel or in response to a failure. The important metric to consider is the difference in boot times excluding the BIOS, which is between 5-7 seconds vs. approximately 2-4 seconds for a virtual machine (VM). This number represents the approximate downtime incurred by a kernel panic when combined with the cost of booting our software stack. We observe that the total cost of warm rebooting either bare metal environment is between 6-7.4 seconds.

After the kernel boots, our software stack usually requires about one second to start up services and modules before resuming a rescued application.

9.1 Basic Applications

To ensure `mini-ckpts` basic correctness, we developed test applications that iterated through hardware registers while setting them to known deterministic patterns in single and multithreaded environments. During execution, we periodically injected a kernel panic. Through over 100 repeated kernel panics, `mini-ckpts` was able to continuously warm reboot the system and continue executing the application without affecting its state.

We next developed a similar test that performed floating point (FPU) calculations on multiple cores designed to only print a failure if the FPU operations (multiply, divide, add, subtract) diverged from their expected value by more than 10^{-5}, to account for FPU rounding errors. Again, `mini-ckpts` was able to protect this FPU test against failures for over 100 consecutive panics during the execution of the same application.

`mini-ckpts` was then evaluated against the `sh` shell, `vi` text editor, and `python` interpreter. While `mini-ckpts` has not been extended to support updating file descriptor seek pointers, it was able to continue executing these applications correctly provided they did not perform extensive file I/O. The `vi` editor did require a command to be blindly typed to reset its terminal display after each panic, since the terminal would not be aware that it needs to refresh after a transparent warm reboot.

9.2 OpenMP Application Performance

We next evaluated `mini-ckpts` with HPC benchmarks from the NAS Parallel Benchmarks (NPB) suite and OpenMP version 3.3 [8]. The results for all environments (8x runs each) are in Figure 1. Bars represent the average runtime per benchmark, excluding intialization, and the error bars indicate the observed minimum and maximum time.

We next discuss the potential for hardware architectures to create performance variability in terms of `mini-ckpts` runtime cost.

9.2.1 NUMA Constraints Imposed by PRAMFS

Naively remapping application memory to PRAMFS may have implications for performance: In Linux, anonymous memory mappings (i.e., heap and stack) may be provided by any free memory. A specialized memory allocator may request memory local to a specific CPU (closest memory controller). PRAMFS, however, is statically allocated to a specific region of physical memory. In non-uniform memory access (NUMA) architectures, this implies that PRAMFS mapped memory regions will have differing latencies between memory accesses on varying cores. We observed that the minimum and maximum runtimes vary by 62% for the CG benchmark when no core pinning was applied to OpenMP threads (Figures omitted due to space constraints). As the scheduler moved threads between cores, the overall runtime of the application varied based on NUMA locality. While CG had the most pronounced effect, the other benchmarks displayed some degree of variance as well.

We were able to confirm that NUMA was responsible for the runtime variance through two experiments:

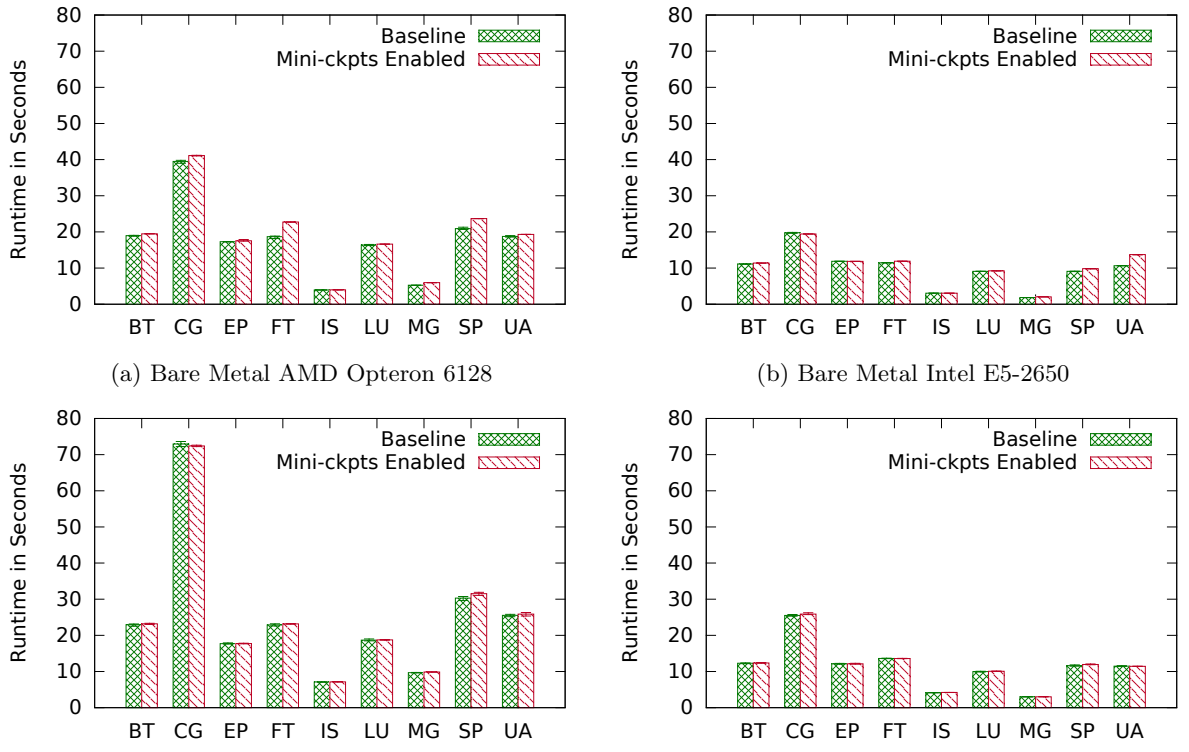
(1) We created a microbenchmark that used `mmap` to map 64MB of memory to our application. In a linear fashion, we then wrote 6GB worth of cumulative writes over this region. Using core pinning, we timed this experiment running on each core. This experiment was repeated for both the AMD and Intel bare metal systems as well as within virtual machines on both. The results of each experiment are shown in Table 5.

Cores	**0-3**	**4-7**	**8-11**	**12-16**
AMD	1.42	2.04	3.25	3.30
AMD VM	3.2 - 3.4			
Intel	0.90		1.12	
Intel VM	0.95			

All Times in Seconds

Table 5: NUMA Microbenchmark Interacting with PRAMFS Memory Mappings

We see locality for PRAMFS, and we also observed that by changing the physical location of the PRAMFS the NUMA localities also move, as expected. Additionally, we compared the runtime of writing to a PRAMFS mapping vs. an `mmap` anonymous memory mappings and found the runtimes had a difference of 2.7% on average (in either direction) for the AMD machine and a 1% difference (in favor of PRAMFS) on the Intel machine. On the virtual machines, we found that regardless of core pinning within the VM, the hypervisor treats each virtual CPU as a thread, which is subject to migration by the hypervisor's scheduler. As shown in Table 5, the AMD VM scheduler experienced performance near the



(a) Bare Metal AMD Opteron 6128

(b) Bare Metal Intel E5-2650

(c) KVM Hypervisor on AMD Opteron 6128 Host

(d) KVM Hypervisor on Intel E5-2650 Host

Figure 1: Runtime Performance of the NPB OpenMP Benchmarks Running with 8 Threads - Core Pinning Applied

its bare metal host's worst NUMA mappings. The Intel VM showed a performance between both its host's best and worst performance as the scheduler migrated the virtual cores during execution.

(2) To isolate the effects of PRAMFS from `mini-ckpts`, we repeated the benchmarks by modifying `mini-ckpts` to only perform PRAMFS remappings, but to not make any further modifications or checkpoints on the process. When combined with core pinning and knowledge of the NUMA latency of each core, we were able to repeat both the best and worst case runtimes for each experiment. Repeated experiments with best-case NUMA mappings and core pinning are shown in Figure 1.

In comparing the two OpenMP experiments with (Figure 1) and without core pinning, we observed that `mini-ckpts` in combination with intelligent physical memory mapping of PRAMFS reduced the AMD host runtimes on average from 26% to 6.9%. The Intel host runtimes were reduced on average from 25% to 5.9%.

9.3 MPI Application Performance

`mini-ckpts` requires customized MPI implementation support to handle lost network connections and buffers on both the sender and receiver. An implementation must also be prepared to handle unexpected failures during system calls by anticipating an `EINTR` return value from system calls that `mini-ckpts` recovered from after a warm reboot. Our prototype implementation, librlmpi, supports these requirements. To provide a meaningful evaluation of the performance of `mini-ckpts` with MPI, we also show performance comparisons between vanilla librlmpi (without `mini-ckpts` enabled) and a mainstream MPI implementation, Open MPI. Additionally, `mini-ckpts` requires remapping all process memory to PRAMFS, so we include an experiment that triggers PRAMFS remapping, but does not activate `mini-ckpts` protection. This experiment provides insight into the effects of PRAMFS memory effects (NUMA) without incurring any `mini-ckpts` overhead. Open MPI currently cannot handle recovery after a `mini-ckpt`, but the relative performance comparison provides the reader with a grasp of librlmpi's performance against a known implementation.

Benchmarks are reported from 8x runs each with bars for minimum/maximum values. We omit detailed results from micro benchmarks in favor of a brief summary due to space.

Figure 2 shows the runtimes of four NPB MPI benchmarks, PENNANT, Clover Leaf over 4 AMD nodes using Gigabit Ethernet. Compared to Open MPI, librlmpi on its own performs better than Open MPI for IS (6% faster). Performance is similar for CG (-1%), EP (0.7%), LU (2%), PENNANT (3%) and is 8% slower for Clover Leaf. The results as both, percent differences and difference in geometric mean (librlmpi in various configurations vs. Open MPI), are shown in Table 6. These results show that librlmpi provides comparable

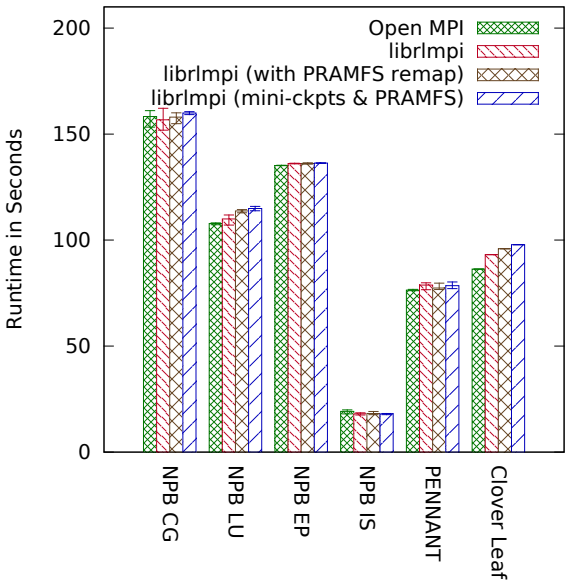


Figure 2: Runtime Comparisons of Various MPI Benchmarks in Differing MPI Stack Configurations

point-to-point message performance to Open MPI at least at our MPI job size when message latency is not a bottleneck to application performance.

Relative to Open MPI, overall average difference of just librlmpi for all benchmarks is 1.1% with a difference in geometric mean of 1.0%. As we enable PRAMFS with librlmpi, the average difference becomes 2.2% with a difference of geometric mean of 2.1%. Finally, with full librlmpi, PRAMFS, and `mini-ckpts` enabled, the average difference becomes 3.1% with a difference of geometric mean of 2.9%.

Table 6 additionally compares librlmpi against itself. Since librlmpi is a prototype demonstrating the capability of `mini-ckpts` to work with an MPI implementation, the overheads of PRAMFS and `mini-ckpts` relative to baseline librlmpi (instead of Open MPI) are now discussed. By subtracting column B from column A, it is observed that on average running an application with data migrated to PRAMFS accounts for 1.1% of the overhead. Further, only considering the costs of `mini-ckpts` by subtracting column C from column B, it is observed that on average `mini-ckpts` accounts for 0.9% of the overhead.

9.3.1 *Injections During MPI Execution*

The performance of the librlmpi prototype is used as a tool to demonstrate `mini-ckpts` functionality rather than its MPI performance. `mini-ckpts` provides protection at a per-node level. We investigated if failures incur a constant cost, independent of the number of processes in a job, as well as, independent of the type of job running. We designed injection experiments for MPI applications to evaluate how `mini-ckpts` scales with MPI. The first experiment involves picking a single node to repeatedly inject kernel failures into. We vary the number of injections seen per run of the application from zero (failure free case) to four. Our second experiment alternates between all nodes each time picking a new node to fail. For instance, in our four process jobs, we first fail node 1, followed by 2, then 3, and finally node 4.

	A	**B**	**C**		
Benchmark	diff. L	diff. L+P	diff. L+P+M	**B-A** P	**C-B** M
NPB CG	-1.0%	-0.2%	1.0%	0.8%	1.2%
NPB LU	2.0%	5.5%	6.5%	3.5%	1.1%
NPB EP	0.7%	0.7%	0.8%	0.0%	0.1%
NPB IS	-6.0%	-5.1%	-6.1%	0.9%	-1.0%
PENNANT	3.1%	1.3%	2.8%	-1.8%	1.5%
Clover Leaf	7.9%	11.2%	13.4%	3.2%	2.2%
Averages	**1.1%**	**2.2%**	**3.1%**	**1.1%**	**0.9%**
Diff.Geo.Mean	**1.0%**	**2.1%**	**2.9%**	**1.0%**	**0.8%**

Table 6: Differences in Runtime & Geometric Mean of MPI Benchmarks with librlmpi(L), PRAMFS(P), and `mini-ckpts` (M) Relative to Open MPI

Figure 3 summarizes our injections during MPI application execution. As each benchmark's failure-free runtime differs, we start the y-axis at 0 and measure only the additional time required when one or more kernel failures are injected. Each MPI application run was conducted with zero and four failures, and each failure was repeated in (1) a scenario where one node receives all of the failures, or (2) the failures were distributed ("alternating") between all nodes. A separate experiment was run for every possible failure count and target. Each experiment was repeated 8 times resulting in error bars that show the observed minimum and maximum runtime, while the points represent the average. We observe a linear runtime increase as the number of injections is incremented for each benchmark.

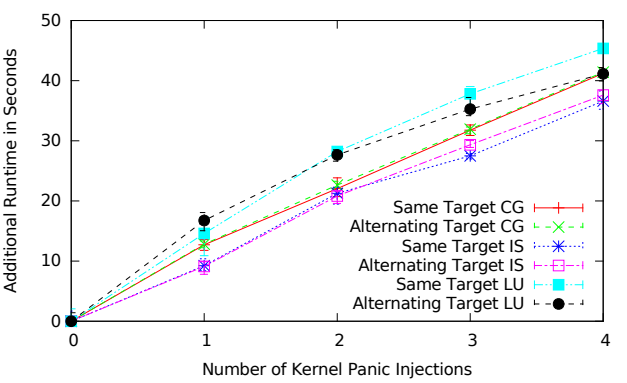


Figure 3: Runtime Costs per Kernel Panic Injection for MPI Applications Demonstrating Linear Slowdown

On average, the additional time increase per injection was between 10-13 seconds, as shown in Table 7. These experiments were run on AMD nodes where the expected total warm reboot time (as shown in Table 4) is previously observed to be 6 seconds, excluding the time to restart an application. We expect that warm reboots will incur an additional slight overhead be-

sides the time taken for the reboot itself since the CPU caches will have been flushed when the operating system restarts. This is similar to a benchmark performing a "warm-up" prior to starting the primary computation. A `mini-ckpts` restart is equivalent to jumping straight into main execution without warming up the cache. The restart times from the MPI injections show that failures increase the application runtime in a linear manner.

# Kernel Panics	1	2	3	4
Time Increase	12.53	11.89	10.76	10.14

All Times in Seconds

Table 7: Average Additional Runtime With Varying Number of Kernel Panics in MPI Applications

9.4 Fault Injections

Fault injections were performed by directly modifying kernel memory from within a kernel module that was triggered by `ioctl` system calls, which identifies the type of injection. Ultimately, we measure `mini-ckpts`' effectiveness based on a successful recovery from an injection that impacted the kernel in such a way that is causing a crash or system hang. For example, some types of kernel injections will either be benign or cause the system to behave incorrectly (go unnoticed without a panic or hang). These types of injections, if they were to occur in the wild, would only be remedied by `mini-ckpts` if the kernel implements a sanity check or bug check that ultimately detects and reacts (by panicking). This is an assumption on our work, but we will point out that — even during the development of `mini-ckpts` — there were several times that `mini-ckpts` misbehaved only to be caught by one of the kernel's safe guards, such as hang detection or file I/O errors. This ultimately caused `mini-ckpts` to reboot the system to a sane state immediately, making kernel development easier than on bare metal without our improvements.

9.4.1 Invalid Pointers

We experimented with pointer injections, either setting a pointer to NULL or switching it to a random value. NULL pointers are the easiest to catch and were protected 100% of the time, since NULL pointer dereferencing is an easily caught exception. Changing a pointer with a non-NULL value had differing outcomes: If the pointer was not a valid address, then it was caught as an exception (leading to a panic), just as easily as a NULL pointer. However, changing the low order bits on a valid kernel pointer typically resulted in a call chain that would randomly jump through the kernel until an invalid reference or sanity check was reached.

During each of the NAS benchmarks we performed the following experiments: (1) We forced a direct call to *panic*. (2) We injected a NULL or invalid pointer to `task_struct` fs member, signal handlers, *parent* member, and/or $files$ member. We were able to detect and recover from each injection. We also scheduled forced panics via a NULL pointer on each core of the system. This test showed that `mini-ckpts` is able to recover from a panic and reboot the system in a valid manner regardless of which OS core it originated on. This was a critical test as it demonstrated the effectiveness of `mini-ckpts` NMI migration protocol during a fault.

9.4.2 Memory Allocation

Memory allocation within the Linux kernel is essential for it to operate properly. An allocation failure when required will force the kernel to panic. `mini-ckpts` does not allocate memory during a panic. However, it is able to induce a kernel panic by exhausting the available memory and waiting for an allocation request to arrive that cannot be fulfilled. We performed allocation experiments repeatedly via `kmalloc` exhausting available memory. Resulting crashes were able to be caught and mitigated through a warm reboot. While this was deliberate, `mini-ckpts` shows promise for mitigating kernel drivers that contain memory leaks on long-running systems. With `mini-ckpts`, a leak could temporarily be coped with by allowing a warm reboot to take place whenever it has exceeded available memory.

9.4.3 Hard Hangs

Some types of faults result in hanging the system, such as an infinite loop in a kernel driver. While Linux is preemptible, a kernel thread may disable preemption to avoid being interrupted during some work. If a fault or bug occurs during this time, it is possible that the CPU will become indefinitely hung. While `mini-ckpts` cannot directly protection against this type of failure, an NMI watchdog may help:

An NMI watchdog within modern hardware periodically sends a non-maskable interrupt to the OS as a forward progress / liveness test. When it fails, the watchdog can directly call a kernel panic (which `mini-ckpts` uses to free the system from its hang and warm reboots). As part of our testing, we injected hangs in regular system calls and interrupt handlers. We were able to successfully detect and recover from all failures.

9.4.4 Soft Hangs

Unlike hard hangs, a soft hang is a software loop that is not making forward progress but is still on a code path that clears the NMI watchdog timer. These types of hangs are much more difficult to detect because the operating system appears to be making progress despite being indefinitely trapped in a loop. While we did not explicitly test for these types of hangs, we did notice that the read-copy-update (RCU) mechanism was able to detect hangs within `mini-ckpts`' file I/O routines taking several minutes to complete during debugging and development.

10. RELATED WORK

Traditional checkpoint/restart of tightly-coupled HPC applications is a well known and researched field. Typically, checkpointing involves saving the state of one

or more processes to stable storage (usually a disk), including the memory, memory layout, open files, threads, and handles to other services provided by the kernel. Checkpointing usually comes in two flavors: (1) system-level checkpointing, such as BLCR [15], which aims to provide transparent checkpoint and restart support to applications via kernel modules without requiring user space application modifications, and (2) user-level checkpointing, such as MTCP [35] and CRIU [17], which also aim to provide transparent or assisted checkpointing to processes without requiring kernel modifications or kernel modules. Although traditional checkpointing methods may vary in how they are implemented or the services they provide, all restore a process to an **earlier** known good state. For long-running applications that are infrequently checkpointed due to the overhead associated with committing a checkpoint, the loss of progress due to a restart from failure may dramatically reduce the efficiency of an application. To mitigate the overhead incurred by checkpointing, a number of optimizations have been studied. These optimizations include: incremental checkpointing [6, 41, 19, 18] which typically saves only modified portions of a program, checkpoint compression [34, 28, 27], and moving incremental checkpoint logic to within kernel [21]. Even with such checkpointing optimizations, when checkpointing, restarts, and rework are modeled to match future exascale systems' expected failure rates, studies show that applications may spend more than 50% of their time in checkpoint/restart/rework phases instead of making forward progress [16, 33, 36]. In these application-based systems, failures that occur within the operating system require an application to rollback and rework as there simply is no other recourse. However, `mini-ckpts` can eliminate unnecessary restart/rework when the application itself remains recoverable, despite a failed kernel.

The idea of an operating system resilient to software errors is not new [7, 29, 12]. The MVS [7] operating system was designed with fault tolerance in mind for both software and hardware errors by requiring that all services be accompanied by an error recovery routine that executes in the event of a failure. This protection increases both the complexity and size of the operating system. In the case of MVS, recovery routines were provided for only 65% of the critical jobs. Despite the efforts of the recovery routines that were provided, in cases were recovery was activated due to a fault, MVS was only successful in preventing an abort 50% of the time [40]. Noting that many kernel failures occur due to bugs within device drivers, the NOOKS [39] framework wrapped calls between the core Linux kernel and device drivers to isolate the effects of one from the other. For cases where bugs are known to exist in drivers, NOOKS provided a transparent solution, but incurs a latency and bandwidth cost due to the wrapping of communication to and from drivers.

The Rio File Cache [13] provides the operating system with a reliable write-back file cache in memory that is capable of surviving warm reboots. Rio guarantees that a file write operation buffered in the kernel during a failure will not be lost, provided that the kernel is able to perform a warm reboot, recover its unperturbed contents, and resume the write operation to stable storage on subsequent reboots. Rio is an example of an independent part of the operating system making guarantees about forward progress regardless of failures by providing a near-zero cost virtual write-through resilience cache.

Protecting the operating system in other ways, such as remote patching in Backdoors [11] or recovering from disk and memory, has been studied, but they have not focused on generalized, transparent high performance application restart in an HPC environment where automatic, low-latency recovery is essential for efficiency [10]. Additionally, work that requires remote intervention or local disk access may not be capable of functioning in circumstances where drivers or kernel support for either the network or disk are not guaranteed. `mini-ckpts` is specialized and superior in this area in that it aims to ensure minimal internal kernel dependencies during fault handling.

Otherworld [14] is perhaps the most similar work to ours in that it attempts to recover applications in the event that a kernel fails. Otherworld maintains a "crash kernel" loaded in separate memory, and it receives control of the system upon failures. Their crash kernel depends on known offsets of data structures within the old kernel to attempt to parse and reconstruct old kernel data structures. This includes a wide variety of traversals and parsing of memory mappings, process structures, file structures, and so forth. Thus, Otherworld depends heavily on an intact and correct kernel state upon failure, which is a significant limitation. The event that caused a kernel to fail may be the effect of a corrupt data structure (i.e., it would continue to exist upon restart) or may create corruption within kernel data structures during the process of crashing. Many corruptions may make Otherworld's reconstruction impossible, while `mini-ckpts` provides non-stop execution without this limitation.

`mini-ckpts` may serve a secondary role beyond reactive fault tolerance for operating systems by providing a means of fast kernel rejuvenation, if used proactively outside the existence of an imminent crash/fault [24]. Outside of OS designs involving non-volatile memory, to the best of our knowledge, this work is the first in its class to provide fast operating system restarts without checkpointing or interrupting an application [9]. Rejuvenating the OS in HPC workloads and virtual machine monitors has been shown to be beneficial in reducing downtime [32, 31].

11. CONCLUSION

Today's operating systems are currently not designed

with fault tolerance in mind, despite the fact that OS memory appears more likely to fail than the remainder of memory. The default mechanism to handle a failure within Linux and commodity OS's is to print an error message and reboot, causing a loss of all unsaved application data on the node and typically triggers a restart for the remainder of the nodes in the application. For these HPC applications, where checkpointing, rollback, and rework are expensive, mitigating an OS crash by allowing warm reboots and recovering applications without data loss can provide a safeguard against memory corruption, system hangs, and other unexpected failures. This work has identified the key points of instrumentation within the Linux kernel required to save the critical state of an application during an OS failure and has provided mechanisms via persistent memory to enable restoration of application data across reboots. We provide an experimental implementation and evaluation of our prototype, `mini-ckpts`, capable of protecting HPC applications from crashes within the kernel by providing non-stop forward progress immediately after a short warm reboot. We show that a `mini-ckpts` enabled kernel can successfully protect an application from computation loss (due to a kernel failure), warm reboot, and transparently restart execution, within 3-6 seconds of a fault occurring. We demonstrate the effectiveness of our work by injecting faults into OpenMP and MPI HPC benchmarks and observe runtime overheads that average between 5% to 6% for OpenMP applications and 3.1% for MPI applications.

12. REFERENCES

[1] Clover Leaf. http://uk-mac.github.io/CloverLeaf/.

[2] OSU MPI micro benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.

[3] PENNANT. https://github.com/losalamos/PENNANT.

[4] Protected and persistent RAM filesystem. http://sourceforge.net/projects/pramfs/.

[5] Top 500 list. http://www.top500.org/, June 2002.

[6] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th Annual International Conference on Supercomputing*, ICS '04, pages 277–286, New York, NY, USA, 2004. ACM.

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

[8] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[9] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[10] M. Baker and M. Sullivan. The recovery box: Using fast recovery to provide high availability in the unix environment. In *In Proceedings USENIX Summer Conference*, pages 31–43, 1992.

[11] A. Bohra, I. Neamtiu, P. Gallard, F. Sultan, and L. Iftode. Remote repair of operating system state using backdoors. In *International Conference on Autonomic Computing (ICAC-04)*, New-York, NY, May 2004. Initial version published as Technical Report, Rutgers University DCS-TR-543.

[12] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – A technique for cheap recovery. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[13] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio File Cache: Surviving operating system crashes. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 74–83, New York, NY, USA, 1996. ACM.

[14] A. Depoutovitch and M. Stumm. Otherworld: Giving applications a chance to survive OS kernel crashes. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 181–194, New York, NY, USA, 2010. ACM.

[15] J. Duell. The design and implementation of Berkeley Lab's Linux Checkpoint/Restart. Tr, Lawrence Berkeley National Laboratory, 2000.

[16] E. N. Elnozahy and J. S. Plank. Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery. *Dependable and Secure Computing, IEEE Transactions on*, 1(2):97–108, Apr. 2004.

[17] P. Emelianov and S. Hallyn. State of criu and integration with lxc. Linux Plumbers Conference, Sept. 2013.

[18] K. B. Ferreira, R. Riesen, R. Brightwell, P. G. Bridges, and D. Arnold. Libhashckpt: Hash-based incremental checkpointing using GPUs. In *Proceedings of the 18th EuroMPI Conference*, Santorini, Greece, September 2011.

[19] K. B. Ferrira, R. Riesen, P. G. Bridges, D. Arnold, and R. Brightwell. Accelerating incremental checkpointing for extreme-scale computing. *FGCS*, 2013.

[20] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 78:1–78:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[21] R. Gioiosa, J. Sancho, S. Jiang, and F. Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 9–9, Nov 2005.

[22] K. Greenan and E. L. Miller. Reliability mechanisms for file systems using non-volatile memory as a metadata store. In *Proceedings of the 6th ACM & IEEE Conference on Embedded Software EMSOFT 06*, pages 178–187, Oct 2006.

[23] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, 33(6):518–528, June 1984.

[24] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. Software rejuvenation: Analysis, module and applications. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 381–390, June 1995.

[25] J. Hulbert. Introducing the advanced XIP file system. In *Linux Symposium*, page 211, 2008.

[26] A. A. Hwang, I. A. Stefanovici, and B. Schroeder. Cosmic rays don't strike twice: Understanding the nature of DRAM errors and the implications for system design. *SIGPLAN Not.*, 47(4):111–122, Mar. 2012.

[27] D. Ibtesham, D. Arnold, P. G. Bridges, K. B. Ferreira, and R. Brightwell. On the viability of compression for reducing the overheads of checkpoint/restart-based fault tolerance. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, 2012.

[28] D. Ibtesham, K. B. Ferreira, and D. Arnold. A study of checkpoint compression for high-performance computing systems. *International Journal of High Performance Computing Applications (IJHPCA)*, 2015.

[29] D. Jewett. Integrity s2: A fault-tolerant unix platform. In *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*, pages 512–519, June 1991.

[30] D. B. Johnson and W. Zwaenepoel. Recovery in distributed systems using asynchronous message logging and checkpointing. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 171–181, New York, NY, USA, 1988. ACM.

[31] K. Kourai and S. Chiba. A fast rejuvenation technique for server consolidation with virtual machines. In *DSN*, pages 245–255. IEEE Computer Society, 2007.

[32] N. Naksinehaboon, N. Taerat, C. Leangsuksun, C. Chandler, and S. L. Scott. Benefits of software rejuvenation on HPC systems. In *ISPA*, pages 499–506. IEEE, 2010.

[33] R. A. Oldfield, S. Arunagiri, P. J. Teller, S. Seelam, R. Riesen, M. R. Varela, and P. C. Roth. Modeling the impact of checkpoints on next-generation systems. In *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, San Diego, CA, September 2007.

[34] J. S. Plank, J. Xu, and R. H. Netzer. Compressed differences: An algorithm for fast incremental checkpointing. *University of Tennessee, Tech. Rep. CS-95-302*, 1995.

[35] M. Rieker, J. Ansel, and G. Cooperman. Transparent user-level checkpointing for the native POSIX thread library for Linux. In *The International Conference on Parallel and Distributed Processi ng Techniques and Applications*, Las Vegas, NV, Jun 2006.

[36] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78(1):012022, 2007.

[37] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: A large-scale field study. In *Proceedings of the* 11^{th} *Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS) 2009*, pages 193–204, Seattle, WA, USA, June 11-13, 2009. ACM Press, New York, NY, USA.

[38] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. *SIGPLAN Not.*, 50(4):297–310, Mar. 2015.

[39] M. M. Swift, S. Martin, H. M. Levy, and S. J. Eggers. Nooks: An architecture for reliable device drivers. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 102–107, New York, NY, USA, 2002. ACM.

[40] P. Velardi and R. K. Iyer. A study of software failures and recovery in the mvs operating system. *IEEE Trans. Computers*, 33(6):564–568, 1984.

[41] S. Yi, J. Heo, Y. Cho, and J. Hong. Adaptive page-level incremental checkpointing based on expected recovery time. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, pages 1472–1476, New York, NY, USA, 2006. ACM.