Evaluating Operating System Vulnerability to Memory Errors

Kurt B. Ferreira, Kevin Pedretti, and Ron Brightwell
Scalable System Software Department
Sandia National Laboratories*
{kbferre|ktpedre|rbbrigh}@sandia.gov

Patrick G. Bridges
Department of Computer Science
University of New Mexico
bridges@cs.unm.edu

David Fiala and Frank Mueller
Department of Computer Science
North Carolina State University
{dfiala|fmuelle}@ncsu.edu

ABSTRACT

Reliability is of great concern to the scalability of extreme-scale systems. Of particular concern are soft errors in main memory, which are a leading cause of failures on current systems and are predicted to be the leading cause on future systems. While great effort has gone into designing algorithms and applications that can continue to make progress in the presence of these errors without restarting, the most critical software running on a node, the operating system (OS), is currently left relatively unprotected. OS resiliency is of particular importance because, though this software typically represents a small footprint of a compute node's physical memory, recent studies show more memory errors in this region of memory than the remainder of the system. In this paper, we investigate the soft error vulnerability of two operating systems used in current and future high-performance computing systems: Kitten, the lightweight kernel developed at Sandia National Laboratories, and CLE, a high-performance Linux-based operating system developed by Cray. For each of these platforms, we outline major structures and subsystems that are vulnerable to soft errors and describe methods that could be used to reconstruct damaged state. Our results show the Kitten lightweight operating system may be an easier target to harden against memory errors due to its smaller memory footprint, largely deterministic state, and simpler system structure.

Keywords

Fault-Tolerance ; Operating Systems ; DRAM Failures

*Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

ROSS '12, June 29, 2012, Venice, Italy

1. INTRODUCTION

Concern is growing in the high-performance computing (HPC) community on the reliability of future extreme scale systems. With systems continuing to grow dramatically in node count and individual nodes also increasing in component count and complexity, large-scale systems are becoming less reliable. In fact, experts are predicting that failure rates may go from the current state of a handful a day [41, 40] to multiple failures an hour [5]. Recent studies have shown soft errors in main memory to be the source of many of these failures [23, 29]. With the predicted increase of memory density on future exascale systems [45] and expected power optimizations such as decreases in supply voltages, the number of these failures is expected to dramatically increase.

Several methods have been developed to address these errors. Approaches include hardware-based techniques, such as single-bit error correction and double-bit detection (SEC-DED) and chipkill codes [13], as well as algorithm-based mechanisms that encode the correction mechanics directly into the application [22, 11, 8]. These hardware-based mechanisms may, however, be insufficient at the elevated failure rates predicted for exascale systems, and most importantly, they may not protect the most important software running on a node - the operating system.

An operating system (OS) resilient to soft errors in memory is key to the scalability of exascale systems for a number of reasons. First, current operating systems are unable to recover from the vast majority of failures. Second, though the typical operating system only occupies a small portion of a system's total physical memory footprint, recent studies show substantially more errors in this region than the remainder of a system's memory [23]. Lastly, future HPC system software will need to continue running in the presence of memory failures if current application-based, forward error recovery mechanisms are to be successful. These forward-error recovery methods are theorized to have lower overheads and less wasted computation than current rollback/recovery mechanisms.

In this work, we investigate the soft error vulnerability of two operating systems used in current and future high-performance computing systems: Kitten, the lightweight kernel developed at Sandia National Laboratories [39], and CLE, a high-performance HPC OS based on the Linux general purpose OS. Our analysis shows that the simpler lightweight kernel may be easier to harden against memory errors, be-

cause of its substantially smaller memory footprint, largely deterministic state, and generally simpler system structure.

2. BACKGROUND

2.1 Current State of Practice

Coordinated checkpoint/restart is the dominant fault tolerance mechanism in high performance computing systems. In current systems, this approach works as follows:

1. Applications periodically quiesce all activity at a global synchronization point, for example a global barrier;
2. After synchronization, all nodes send some fraction of application and system state, generally comprising most of system memory, over the network to dedicated I/O nodes;
3. These I/O nodes store the received checkpoint information data to stable storage, currently hard disk-based storage;
4. In the event of application crash, the stored checkpoint can be used to restart the application at a prior known-good state.

The continued dominance of this technique rests on a number of key assumptions regarding failures that have thus far remained true:

1. Application state can be saved and restored much more quickly than a system's mean time to interrupt (MTTI);
2. The hardware and upkeep (e.g. power) costs of supporting frequent checkpointing are a modest portion (currently perhaps 10-20%) of the system's overall cost; and
3. System faults that do not crash (fail-stop) the system, such as so-called "soft errors", are very rare.

In an environment where failures are common, traditional checkpoint/restart has been shown to be inappropriate for large-scale systems [41, 3, 5, 19]. Additionally, checkpoint/-restart is problematic when dealing with non-crash failures. In particular, checkpoint/restart *preserves* the impact of failures that corrupt application state. Addressing this problem requires application developers to either restart the application from scratch or analyze the contents of their checkpoints looking for one prior to when the fault that corrupted application state occurred.

Because of this limitation, there is significant effort underway within the community to develop *forward-error recovery* methods for application fault tolerance [19, 16, 7]. These methods deal with faults by correcting lost or incorrect state rather than restarting an application from a previously saved state. This approach avoids the wasted power and work of rollback/recovery methods like checkpointing and typically have significantly lower overheads.

2.2 DRAM Failures

Recent studies have shown DRAM errors in main memory to be the most common source of failures on today's HPC platforms [23, 29]. The prevalence of these DRAM errors is related to the fact that typical large scale systems contain tens to hundreds of thousands of DRAM modules. A combination of the quantity and density of the information stored makes these modules particularly susceptible to faults. Moreover, with expected power optimizations, such as decreased supply voltages and increases in memory densities, the number of DRAM errors is expected to increase for future exascale systems [45].

To address these faults, current HPC systems typically include some form of error correction. The most common memory resilience scheme has the memory controller write additional checksum bits on each block of data. The memory controller then uses these checksum bits to detect and correct DRAM errors. Single-symbol Error Correction and Double-symbol Error Detection (SEC-DED) schemes allow systems to recover from the simplest memory failures and at least detect more complex (and less frequent) ones; or more complex chipkill-based codes [13] that allow a system to tolerate an entire DRAM chip failure at the cost of reduced performance and increased energy usage.

Uncorrectable DRAM errors, errors to two or more bits, are becoming increasingly common in systems with SEC-DED memory protection [42], with these errors occurring in up to 8% of DIMMs per year. For an exascale class system, this translates to multiple uncorrectable errors per hour. Such errors generally result in a machine check exception being delivered to the operating system, which then typically logs the error, and either kills the application to which the memory location belongs, or reboots the system if the error resides in a critical portion of the operating system's address space [27].

As stated earlier, though the typical operating system occupies a very small portion of the system's total physical memory, errors within the operating system's address space are much more likely to occur than errors within the remainder of memory [23]. Therefore, techniques to address these errors at the system level are critical to the scalability of exascale systems.

3. APPROACH

The advantages described thus far in this paper provide a compelling reason to evaluate an HPC operating system's vulnerability to memory errors. In this evaluation, we consider two operating systems of the type we expect to see on an exascale class system. The first is the Kitten lightweight kernel [39] developed by Sandia National Laboratories. The second is a variant of the Linux general-purpose operating system, called the Cray Linux environment.

Kitten is a special-purpose, limited-functionality OS designed for use on the compute nodes of massively parallel supercomputers. Its code base is derived from Linux, but is modified to minimize kernel-level functionality to only that needed for a set of mission-critical HPC applications and moves as much as possible into user-space. Kitten is similar to previous lightweight kernels (LWK) such as SUNMOS, Puma, Cougar, Catamount, and IBM's CNK [1]. Kitten, however, distinguishes itself from these prior LWKs by providing a combination of a Linux-compatible user environment [1], a more modern and extensible code base, and a virtual machine monitor capability via the Palacios virtual machine monitor [32] which allows full-featured guest operating systems to be loaded on-demand and at very low overhead [28].

The Cray Linux Environment (CLE) is Cray's scalable operating system for their XT line of supercomputers. CLE is

based on the Linux general-purpose operating system with the addition of a number of optimizations to improve scalability. These optimizations include: enhancements to memory management, improved out-of-memory handling, and modifications for decreased OS jitter.

In this work, we will consider vulnerability to three types of common memory failures:

- Detected and corrected single-bit errors
- Detected but uncorrectable multi-bit errors
- Undetected "silent" data corruption

While fully protecting against each of these error types would be ideal, in many cases, the cost of doing so would far outweigh the benefit. Our goal is to identify the highest-impact opportunities for improving an OS's resilience to memory errors.

Our evaluation will proceed as follows. First, for each OS, we will look at its complexity and how that complexity changes as a function of time. Our metric for complexity will be Source Lines Of Code (SLOC) count [12]. This metric gives us a rough measure of how difficult constructing and managing memory error mitigation methods will be. Next, we compare the memory footprints of the two operating systems, outlining how these footprints may change as an application progresses. Lastly, we breakdown the vulnerability of an OS on a per-subsystem basis, enumerating the subsystems' critical state (state that must be free of errors). Additionally, for this critical state we describe possible failure mitigation strategies.

4. RESULTS

4.1 Source Lines of Code

The Linux kernel has been enormously successful in attracting developers and users over its twenty year history. Due to this large development community and strong hardware support, Linux has also been successful in attracting HPC developers and is widely used within the community. Figure 1(a) plots the growth of the full Linux kernel codebase in terms of source lines of code (SLOC), tracking its growth from approximately 120K SLOC in 1994 to its present size of over 10M SLOC. As the figure shows, the majority of the codebase consists of drivers. However, as shown in the right graph of Figure 1, non-driver core kernel code is also considerable and is growing rapidly. The current version of Linux, version 3.3, consists of approximately 350K SLOC in the core x86 architecture port (/kernel, /mm, and arch/x86 directories).

The Kitten codebase, in contrast, is currently a total of 246K SLOC, which drops to 66K SLOC once the Infiniband drivers and associated Linux driver support code are removed. Kitten's core kernel code for the x86 architecture port is 30K SLOC, which is an order of magnitude smaller than the corresponding subset of Linux. This suggests that Kitten is considerably less complex than Linux, and will be easier to harden against memory errors.

4.2 Memory Footprint Comparison

Figure 2 compares the physical memory layouts used by Kitten and Linux. The primary difference between the two is that Kitten explicitly partitions memory into two regions, one for kernel memory and another for user-space applications, while Linux uses a unified page pool and dynamically assigns pages to different roles as needed. Kitten's kernel memory footprint has a fixed upper limit (currently 64 MB) that does not change during runtime, while Linux's footprint changes over time and can grow to the maximum size of physical memory.

Each user-space process on Kitten requires three pages of kernel memory to store task and address space structures, as well as a static amount of kernel memory to store the application's page tables. When using 2 MB pages on the x86 architecture, approximately 8 KB of page table memory is needed for each gigabyte of application memory. Linux has similar per-process kernel memory requirements, with the addition of the pages in the page cache being used by the process. Kitten does not have a page cache. Additionally, Linux uses the 4 KB page size by default, resulting in more kernel memory being used for page tables (2 MB per GB of application memory). Libraries such as libhugetlbfs and recent transparent large page support in Linux are making it easier for applications to use large page sizes, with the caveat that memory fragmentation over time causes significant issues.

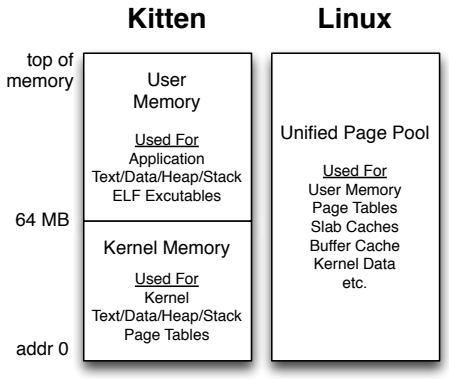
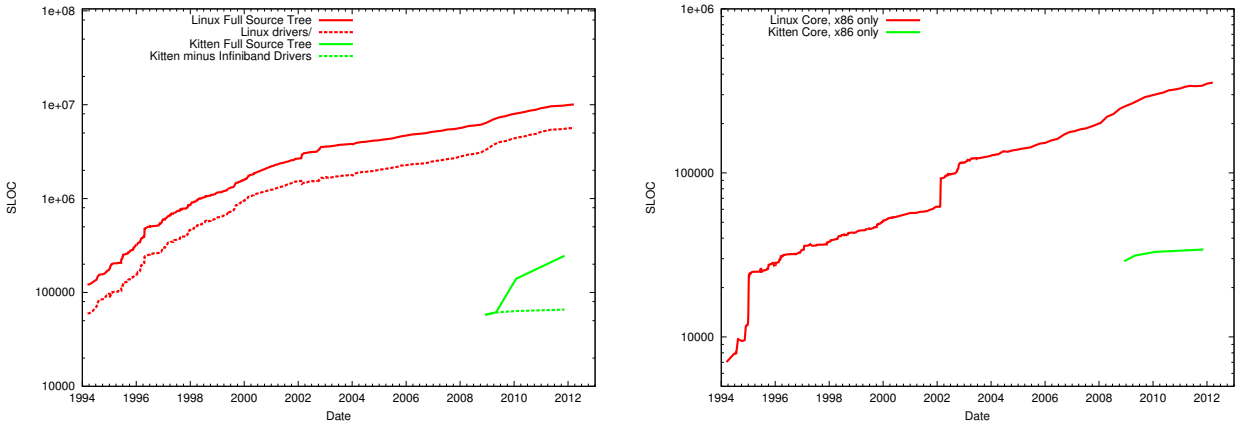


Figure 2: Physical memory layout of Kitten and Linux.

While a standard Linux kernel can grow to the full extent of the physical memory on the machine, typically the size is much smaller. In fact, CLE has a number of memory usage optimizations that limit memory footprint size. Specifically, CLE limits the size of the page cache using an I/O forwarding technique that avoids caching of file reads and writes. Figure 3 shows a comparison of the Kitten and CLE footprints. For Kitten, memory partitioning limits total kernel size to 64MB. For the CLE, we show the average kernel size measured using the smem [46] memory tracking tool while running the LAMMPS [38] molecular dynamics code from Sandia National Laboratories. From the figure, we see that worst case Kitten OS size is more than an order of magnitude smaller than the average case from the CLE. Kitten's smaller and deterministic footprint generally means simpler methods to protect and correct this state due to DRAM errors.

4.3 Major Kernel Subsystems

This section examines several kernel subsystems that ex-



(a) Full Tree

(b) Core Kernel Code, x86 Port Only

Figure 1: Comparison of Linux Kernel and Kitten Kernel source lines of code (SLOC).

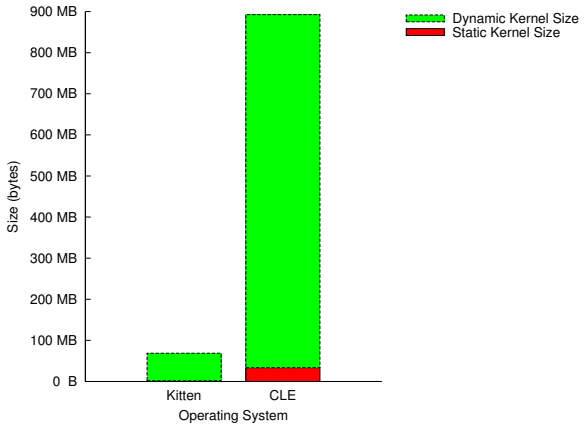


Figure 3: Comparison of the worst case Kitten static and dynamic kernel size to the average case measured on CLE. The average CLE memory footprint is an order of magnitude larger then the worst case for Kitten.

ist in both operating systems, and discusses techniques that could be used to harden them against memory errors. The subsystems are discussed in the order of their kernel memory footprint in the Kitten kernel. This analysis captures the vast majority of Kitten's kernel memory footprint, and is representative of the baseline kernel-level functionality needed to support highly-scalable HPC applications.

4.3.1 Page Table Memory

Both Kitten and Linux store page tables in kernel memory. The amount of page table memory used varies depending on the page size used: 4 KB pages require 2 MB of page table memory per gigabyte of application memory, 2 MB pages require 8 KB per gigabyte, and 1 GB pages require 8 bytes per gigabyte. In general, Kitten is always able to use the larger page sizes for application memory due to its segment-based and static memory allocation policy. Recent Linux kernels attempt to use large page sizes when possible, but memory fragmentation can limit the usefulness of this optimization.

Both OSs consider page table memory errors as fatal, either killing the affected application or the entire node. However, Kitten's deterministic mapping of virtual to physical addresses would make it straightforward to recreate the corrupted page table memory contents from the base physical address and length information stored in the address space region object. This would not work on Linux due to its demand paging scheme, where unpredictable physical addresses are assigned to virtual addresses at runtime. Extra redundant state would need to be stored, and furthermore it may be difficult or impossible to tell which page table values have become corrupted if hardware notification is not provided.

4.3.2 Physical Memory Management

As described in Section 4.2, Linux uses a unified physical page pool. Linux maintains a memory map array, with one entry for each page of physical memory, to track the current state of each page frame in the system. Each entry in the table is 56 bytes, requiring 14 MB of overhead per GB of physical memory (1.4%).

Kitten does not maintain a memory map array. Instead, it maintains a free list of physical memory segments, where each segment consists of a physically contiguous set of pages with identical type (e.g., allocation status, memory type, associated NUMA node). Kitten's segment list typically holds 10–100 entries for a high-end NUMA system, requiring less than 4 KB of kernel memory. Kitten's physical memory tracking scheme would not work well for a general-purpose kernel, but is a good match for its target workloads where applications are allocated large contiguous regions of physical memory that are not demand paged.

As with page table memory, memory errors to the physical memory tracking data structures are considered fatal. In this case, however, there is no easy way to recreate the corrupted state. Instead, additional state of some form would have to be maintained, such as software-maintained ECC bits or redundant copies.

4.3.3 Dynamic Kernel Memory

Both OS kernels provide a mechanism for kernel subsystems and device drivers to allocate dynamic memory, similar to malloc() at user-level. Kitten implements this functionality using a buddy system memory allocator that covers the kernel memory portion of the physical address space (by default 64 MB). To avoid wasting memory due to over-allocation, Kitten uses a minimum block size of 32 bytes, which results in approximately 2 MB of buddy allocator state (one bit per 32 byte block). Kitten was profiled on an 8-core Intel system running an 8-thread OpenMP benchmark and found to use a maximum of 45 KB of dynamically allocated kernel memory.

Linux implements dynamic kernel memory allocation via a slab cache [6], which allocates physical memory from a buddy allocator that covers all of physical memory. The buddy allocator uses a 4 KB minimum block size, resulting in approximately 256 KB of overhead per gigabyte of physical memory. Each slab cache requires a small state tracking structure of approximately 128 bytes plus 32 bytes per NUMA node. As an example, a Cray XE6 compute node running CLE 4.0.36 (Linux 2.6.32.45) maintains 130 slab caches of various sizes.

In addition to all of the memory allocator data structures being assumed to be reliable (buddy allocator state, slab cache info), each block of memory allocated has a small header at its start storing the size of the block and where it should be returned when freed (16 bytes for Kitten). This data would need to be protected from memory errors somehow, possibly by an ECC-like code or by storing redundant copies of the header. Alternatively, the caller could be made aware of the header so that it might try to protect it.

4.3.4 Address Spaces and Tasks

At its heart, Kitten's main purpose is to bootstrap user-space address spaces and tasks (threads and processes) and then get out of the way. Both address spaces and tasks are tracked by kernel-level state structures. Kitten's task structure is 8 KB in size and includes the task's kernel-level stack. Kitten's address space structure is around 800 bytes in size. Both structures are almost entirely self-contained, with only two pointers to additional data structures. Kitten's address space structure points to a list of virtual memory regions, of which there are usually four for a typical application address space: text, data, heap, and stack.

Linux has similar, but more complex task and address space structures. For example, the Linux task structure has over 160 fields, compared to 23 fields for Kitten. The obvious reason for this large difference is the additional functionality that Linux provides. However, much of this is not useful for HPC workloads, and simply increases the effort needed to understand and protect the codebase.

4.3.5 Kernel Entry and Exit

Kernel entry and exit occurs through well-defined interfaces. Both Linux and Kitten route all interrupts through a small assembly stub, which saves the necessary state and then calls the appropriate higher-level handler. Similarly, when applications make system calls, the kernel is entered through a common routine, which then redirects through a table to the appropriate handler.

This structure could potentially be leveraged to do coarse-grained kernel memory error detection and correction. At each kernel entry and exit, the entire kernel memory space could be checksummed to ensure that no kernel data was silently corrupted. This is straightforward on Kitten due to its contiguous kernel memory region. On Linux, kernel memory and application memory is interleaved both in physical memory and in the kernel's virtual address space, making the checksum process more difficult but still possible.

Clearly, this approach would have high overhead when invoked. However, HPC applications typically do not make many system calls, and could benefit from the increased protection from memory errors. Additionally, it would eliminate the need to protect each individual kernel data structure, reducing memory overhead.

4.4 Page Retirement

An additional technique that applys to all of the kernel subsystems discussed thus far is page retirement [23]. In this scheme, the OS monitors the memory errors corrected by hardware and uses this information to predict which memory pages are likely to fail soon. Kernel data structures using these pages can then be migrated to more stable memory pages or discarded if appropriate. Recent versions of Linux can already use this technique to discard clean page cache pages that have experienced an uncorrectable memory error.

Kitten and Linux are both written in C, which makes migrating kernel data structures difficult since it is difficult to determine which other structures point to the data being moved. Furthermore, it is difficult to determine which kernel-level data structures are using a given page. Therefore, both OSs would require heavy modification in order to take advantage of this technique. In this regard, Kitten's smaller codebase could potentially be an advantage.

5. RELATED WORK

Resiliency and fault-tolerance has been identified by the Department of Energy and Department of Defense as one of the key fundamental challenges of extreme-scale computing. The majority of the work in this active research area has focused solely on the application and ignored the operating and runtime systems, which is the focus of this work. Essentially all of these approaches attempt to improve the performance of checkpoint/restart as it is the most widely used mechanism for fault-tolerance today. To the best of our knowledge, the only work similar to our resilient operating and runtime systems work is in the context of reliability for hostile environments, such as outer space and high radiation environments [43, 36, 34, 37, 33]. These methods typically have high runtime overheads [20] and it is unclear if they are appropriate for HPC.

In addition to the HPC application-based methods, a small handful of researchers have been focusing on designing fault-tolerant userspace libraries for HPC systems that applications can use to construct algorithm-based resilience. In each of these research areas is an underlying assumption that the operating and runtime systems are resilient to failures or if not, an expensive restart of the OS must be done. In the remainder of this section, we briefly describe each of these approaches and discuss their potential benefits and costs.

5.1 High-speed Storage for Checkpoint/Restart

Checkpointing to local disk and flash memory systems has periodically been proposed to speed up checkpoint/restart

systems by placing large amounts of high-speed storage near the data that must be checkpointed. Actually deploying large amounts of local non-volatile storage in an exascale system is potentially very challenging. Local disk-based storage has traditionally been avoided because of the increased failures it causes, for example. Upcoming non-volatile phase change PCRAM, resistive RRAM devices, and modern NAMD and NOR flash technologies provide high bandwidth and reliability, but are potentially very expensive. Unless their cost per bit rivals that of disk, using such technologies for checkpoint/restart purposes would result in checkpointing hardware that makes up a much larger portion of the system cost. Additionally, write durability issues may require periodically replacing all flash memory in the system, further impacting total costs.

5.2 Asynchronous Checkpointing and Message Logging

Another approach that has been suggested to improve the performance of checkpointing systems is uncoordinated or asynchronous checkpointing [2, 25, 26]. These methods typically checkpoint and restore from local storage without the synchronization used by coordinated checkpointing. To support a node restoring from a local asynchronous checkpoint, nodes in this approach keep a log of recent messages that they have sent. When a node restores from a previous checkpoint, it can then replay reception of messages using remote nodes' logs.

While this approach can increase checkpointing performance, logging increases the latency of messaging operations and potentially takes significant amounts of memory on a node. Finally, asynchronous checkpointing approaches can result in cascading rollbacks; recent work attempts to bound the amount of rollback that may be necessary [21], but also places non-trivial limits on application behavior. Lastly, thus far there has been little work examining the performance of a general message logging approach at the scales one might expect to see at exascale.

5.3 Other Checkpointing Systems

Memory-based checkpointing [35, 44] uses the memory of a remote machine to checkpoint node state. Unless node memory is primarily read-only (in which case RAID 5-like techniques can be used), this approach doubles the memory demands of an application. Since memory is regarded as a key budget and power constraint in exascale systems, the benefits of these techniques are unclear.

Multi-level checkpointing [31] is a library-based approach for controlling checkpointing to multiple storage targets, including memory-based checkpoints, local checkpoint storage, and remote checkpoints, into a single system. Because of this, it shares some of the advantages and disadvantages of memory-based checkpointing and local storage techniques. Unlike these techniques, however, multi-level checkpointing has the flexibility to choose between multiple levels of storage based on system design parameters, making it a promising technique for exascale systems.

Finally, recent studies have looked at the benefits and costs of combining replication with traditional checkpoint/restart [19, 17, 15]. These studies seek to find the "break-even" points for replication, or the point where this replication approach uses fewer resources then traditional checkpoint/restart alone. In contrast to the other methods described thus far in this section, since replication typically duplicates not only the application processes but also a subset of the OS instances, errors with the operating and runtime system can be handled.

5.4 Fault Tolerant Userspace Libraries

In contrast to the checkpointing work described above, a number of researchers are investigating constructing libraries that are tolerant to certain kinds of faults. The idea being that the applications use these libraries to construct application-specific fault tolerance mechanisms, typically termed algorithm-based fault tolerance (ABFT) [22]. These ABFT techniques typically require a fault-tolerant message passing environment. There have been a number of these resilient message passing libraries based on MPI, including; FT-MPI [24, 18], AMPI [10], MPI/FT [4], and C^3 [9]. The differences between these libraries is beyond the scope of this work, but each of these libraries allows for an application to continue operating in the presence of faults, possibly in a degraded mode, and it is left up to the application to ensure the result is correct.

5.5 Current Operating System Memory Error Handling

OS-level handling of DRAM faults has generally been either very limited or used very heavyweight solutions. Linux and other operating systems, for example, provide low-level techniques for handling, logging, and notifying the application of such errors [27]. These techniques generally terminate the application or OS kernel, and potentially invoke higher-level recovery systems based on, for example, checkpointing or redundancy. Some systems have attempted to provide additional protection against memory faults both on CPUs [14] and GPUs [30], though with substantial cost.

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented a preliminary evaluation of operating system vulnerability to DRAM failures, a common error in current and future extreme-scale systems. Hardening system software to this class of errors will be critical for the success of emerging fault-tolerance methods. This work focused on two HPC operating systems; Kitten, the lightweight operating system developed at Sandia National Laboratories and the Cray Linux Environment, a HPC variant of the Linux operating system. Each of these OSs represents an OS construction methodology currently used in HPC. For each OS, we present the complexity of each OS in terms of the metric SLOCCount, examine the memory footprint, and evaluate vulnerability on a per subsystem basis. Where critical state is found, state that must be protected from DRAM errors, we outline mitigation methods that can used. Overall, these results suggest hardening the Kitten lightweight kernel to to be more tractable due to its smaller and deterministic state in comparison to Linux.

While this preliminary analysis shows there is promise in this idea, more work is clearly needed. For example, a detailed analysis of the mitigation techniques, outlining both the space and performance overheads is need to decide which methods are ideal. Additionally, hardening system software to failures beyond those that occur in system RAM will be key to scalability of extreme-scale systems. Lastly, evaluating the hardened OSs and system services to errors will be key to outlining this work's overall merit.

7. REFERENCES

[1] Adiga, N., and et al. An overview of the BlueGene/L supercomputer. In *Supercomputing, ACM/IEEE 2002 Conference* (nov. 2002), p. 60.

[2] Ahn, J. 2-step algorithm for enhancing effectiveness of sender-based message logging. In *SpringSim '07: Proceedings of the 2007 spring simulation multiconference* (2007), pp. 429–434.

[3] Amarasinghe, S., and et al. Exascale software study: Software challenges in extreme scale systems. `http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/ECSS%20report%20101909.pdf`, Sept. 2009.

[4] Batchu, R., Dandass, Y. S., Skjellum, A., and Beddhu, M. MPI/FT: A model-based approach to low-overhead fault tolerant message-passing middleware. *Cluster Computing 7*, 4 (Jan. 2004), 303–315.

[5] Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hill, K., Hiller, J., Karp, S., Keckler, S., Klein, D., Kogge, P., Lucas, R., Richards, M., Scarpelli, A., Scott, S., Snavely, A., Sterling, T., Williams, R. S., and Yelick, K. Exascale computing study: Technology challenges in achieving exascale systems. `http://www.science.energy.gov/ascr/Research/CS/DARPAexascale-hardware(2008).pdf`, Sept. 2008.

[6] Bonwick, J., and Adams, J. Magazines and vmem: Extending the slab allocator to many CPUs and arbitrary resources. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2001), USENIX Association, pp. 15–33.

[7] Bridges, P., Hoemmen, M., Ferreira, K. B., Heroux, M., Soltero, P., and Brightwell, R. Cooperative application/os DRAM fault recovery. *Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids in conjunction with the Euro-Par Conference, Lecture Notes in Computer Science* (2011), –.

[8] Bronevetsky, G., and de Supinski, B. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the 22nd Annual International Conference on Supercomputing* (New York, NY, USA, 2008), ICS '08, ACM, pp. 155–164.

[9] Bronevetsky, G., Marques, D., Pingali, K., and Stodghill, P. Collective operations in application-level fault-tolerant MPI. In *Proceedings of the 17th annual international conference on Supercomputing* (New York, NY, USA, 2003), ICS '03, ACM, pp. 234–243.

[10] Chakravorty, S., Mendes, C., and KalÃl', L. Proactive fault tolerance in mpi applications via task migration. *Strategy 4297* (2006), 485âĂŞ496.

[11] Chen, Z., and Dongarra, J. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International* (April 2006).

[12] David A. Wheeler. Sloccount. `http://www.dwheeler.com/sloccount`, March 1 2012.

[13] Dell, T. J. A white paper on the benefits of chipkill-correct ECC for PC server main memory. IBM Microelectronics Division, Nov. 1997.

[14] Dopson, D. SoftECC: A system for software memory integrity checking. Master's thesis, Massachusetts Institute of Technology, September 2005.

[15] Elliot, J., Kharbas, K., Fiala, D., Mueller, F., Ferreira, K., and Engelmann, C. Combining partial redundancy and checkpointing for HPC. In *International Conference on Distributed Computing Systems* (Los Alamitos, CA, USA, June 2012), IEEE Computer Society Press, pp. 1–11. [to appear].

[16] Engelmann, C., and Geist, G. A. A. Super-scalable algorithms for computing on 100,000 processors. In *Lecture Notes in Computer Science: Proceedings of the 5^{th} International Conference on Computational Science (ICCS) 2005, Part I* (Atlanta, GA, USA, May 22-25, 2005), vol. 3514, Springer Verlag, Berlin, Germany, pp. 313–320.

[17] Engelmann, C., Ong, H. H., and Scott, S. L. The case for modular redundancy in large-scale high performance computing systems. In *Proceedings of the 8^{th} IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2009* (Innsbruck, Austria, Feb. 16-18, 2009), ACTA Press, Calgary, AB, Canada, pp. 189–194.

[18] Fagg, G. E., Angskun, T., Bosilca, G., Pjesivac-Grbovic, J., and Dongarra, J. Scalable fault tolerant mpi: Extending the recovery algorithm. In *PVM/MPI* (2005), B. D. Martino, D. Kranzlmüller, and J. Dongarra, Eds., vol. 3666 of *Lecture Notes in Computer Science*, Springer, pp. 67–75.

[19] Ferreira, K., Riesen, R., Stearley, J., III, J. H. L., Oldfield, R., Pedretti, K., Bridges, P., Arnold, D., and Brightwell, R. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage, and Analysis, (SC'11)* (Nov 2011).

[20] Fiala, D., Ferreira, K. B., Mueller, F., and Engelmann, C. A tunable, software-based DRAM error detection and correction library for HPC. In *Lecture Notes in Computer Science: Proceedings of the European Conference on Parallel and Distributed Computing (Euro-Par) 2011: Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids* (Bordeaux, France, Aug 2011), Springer Verlag, Berlin, Germany.

[21] Guermouche, A., Ropars, T., Brunet, E., Snir, M., and Cappello, F. Uncoordinated checkpointing without domino effect for send-deterministic message passing applications. In *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium* (May 2011).

[22] Huang, K.-H., and Abraham, J. A. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers C-33*, 6 (June 1984).

[23] Hwang, A. A., Stefanovici, I. A., and Schroeder, B. Cosmic rays don't strike twice: understanding the nature of DRAM errors and the

implications for system design. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS '12, ACM, pp. 111–122.

[24] Inovative Computing Laboratory. FT-MPI. `http://icl.cs.utk.edu/ftmpi`, March 1 2012.

[25] Jiang, Q., and Manivannan, D. An optimistic checkpointing and selective approach for consistent global checkpoint collection in distributed systems. In *Proceedings of the 2007 IEEE International Parallel and Distributed Processing Symposium* (Mar. 2007).

[26] Johnson, D. B., and Zwaenepoel, W. Recovery in distributed systems using asynchronous and checkpointing. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing* (1988), pp. 171–181.

[27] Kleen, A. mcelog: memory error handling in user space. In *Proceedings of Linux Kongress 2010* (Nuremburg, Germany, September 2010).

[28] Lange, J. R., Pedretti, K. T., Hudson, T., Dinda, P. A., Cui, Z., Xia, L., Bridges, P. G., Gocke, A., Jaconette, S., Levenhagen, M., and Brightwell, R. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *IPDPS'10* (2010), pp. 1–12.

[29] Li, S., Chen, K., Hsieh, M.-Y., Muralimanohar, N., Kersey, C. D., Brockman, J. B., Rodrigues, A. F., and Jouppi, N. P. System implications of memory reliability in exascale computing. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2011), SC '11, ACM, pp. 46:1–46:12.

[30] Maruyama, N., Nukada, A., and Matsuoka, S. A high-performance fault-tolerant software framework for memory on commodity GPUs. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on* (april 2010), pp. 1 –12.

[31] Moody, A., Bronevetsky, G., Mohror, K., and Supinski, B. R. d. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (Washington, DC, USA, 2010), SC '10, IEEE Computer Society, pp. 1–11.

[32] Northwestern University. Palacios: An os independent embeddable vmm. `http://v3vee.org/palacios`, March 10 2012.

[33] Oh, N., Shirvani, P., and McCluskey, E. Control-flow checking by software signatures. *Reliability, IEEE Transactions on 51*, 1 (mar 2002), 111–122.

[34] Oh, N., Shirvani, P., and McCluskey, E. J. Error detection by duplicated instructions in super-scalar processors. *Reliability, IEEE Transactions on 51*, 1 (mar 2002), 63–75.

[35] Plank, J. S., Kim, Y. B., and Dongarra, J. J. Algorithm-based diskless checkpointing for fault tolerant matrix operations. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers* (Pasadena, CA, USA, June 1995), Los Alamitos, CA, USA : IEEE Comput. Soc. Press, 1995, pp. 351–360.

[36] Rebaudengo, M., Reorda, M., Violante, M., and Torchiano, M. A source-to-source compiler for generating dependable software. In *Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE International Workshop on* (2001), pp. 33–42.

[37] Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., and August, D. I. SWIFt: Software implemented fault tolerance. In *Proceedings of the international symposium on Code generation and optimization* (Washington, DC, USA, 2005), CGO'05, IEEE Computer Society, pp. 243–254.

[38] Sandia National Laboratories. The LAMMPS molecular dynamics simulator. `http://lammps.sandia.gov`, April 2010.

[39] Sandia National Laboratory. Kitten lightweight kernel. `https://software.sandia.gov/trac/kitten`, March 10 2012.

[40] Schroeder, B., and Gibson, G. A. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN2006)* (June 2006).

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

[42] Schroeder, B., Pinheiro, E., and Weber, W.-D. DRAM errors in the wild: a large-scale field study. *Communications of the ACM 54* (February 2011), 100–107.

[43] Shirvani, P., Saxena, N., and McCluskey, E. Software-implemented EDAC protection against SEUs. *Reliability, IEEE Transactions on 49*, 3 (sep 2000), 273 –284.

[44] Silva, L. M., and Silva, J. G. An experimental study about diskless checkpointing. In *24th EUROMICRO Conference* (Vasteras, Sweden, August 1998), IEEE Computer Society Press, pp. 395 – 402.

[45] Simon, H. Exascale challenges for the computational science community. Tech. rep., Lawrence Berkeley National Laboratory and UC Berkeley, Oct. 2010.

[46] SMEM. Memory reporting tool. `http://www.selenic.com/smem/`, March 1 2012.