FlipSphere: A Software-based DRAM Error Detection and Correction Library for HPC

David Fiala and Frank Mueller
Dept. of Computer Science
North Carolina State University
Email: mueller@cs.ncsu.edu

Kurt B. Ferreira
Scalable System Software
Sandia National Laboratories
Email: kbferre@sandia.gov

***Abstract*—Proposed exascale systems will present considerable resiliency challenges. In particular, DRAM soft-errors, or bit-flips, are expected to greatly increase due to much higher memory density of these systems. Current hardware-based fault-tolerance methods cannot cope by itself with the expected soft error frequency rate. As a result, additional software is needed to address this challenge.**

We introduce FlipSphere, a tunable, transparent silent data corruption detection and correction library for HPC applications that is first in its class to use hardware accelerators such as the Intel Xeon Phi MIC to increase application resiliency. FlipSphere provides comprehensive silent data corruption protection for application memory by implementing on-demand page integrity verification coupled with software-based error correcting codes that allow for automatic error recovery in the event of memory failures. We investigate the trade-off of hardware resources for resilience and find that up to 90% of memory may be guarded with error detection at a 40% performance overhead.

I. Introduction

With the increased density and power concerns in modern computing chips, components are shrinking, heat is increasing, and hardware sensitivity to outside events is growing. These variables, combined with the extreme number of components expected to make their way into computing centers as our computational demands expand, are posing a significant challenge to the design and implementation of future extreme-scale systems. Of particular interest are soft errors in memory (spontaneous bit flips) that manifest themselves as silent data corruption (SDC). SDC is of great importance to the reliability of these systems due to its ability to render results invalid in scientific applications without detection of such corruption.

Silent data corruption can occur in many components of a computer system including the processor, cache, and memory due to radiation, faulty hardware, and/or lower hardware tolerances. While cosmic particles are one source of concern, another growing issue resides within the circuits themselves, due to miniaturization of components. As components shrink, heat becomes a design concern, which in turn leads to lower voltages in order to sustain the growing chip density. Lower component voltages result in a lower safety threshold for the bits that they contain, which increases the likelihood of an SDC occurring. Further, as densities continue to grow, any event that upsets chips (i.e., radiation) is more likely to flip bits.

Current systems use memory with hardware-based ECC that is capable of correcting single bit errors and detecting double bit errors [1] within a region of memory (typically a cache line). Errors in current systems that result in three or more bit flips will produce undefined results including silent data corruption, which may produce invalid results without warning. While the frequency of single and double bit errors is known (8% of systems will incur correctable errors while 2%-4% of will incur uncorrectable errors [2]), the frequency of higher bit errors is still an open research question, and GPUs are known to be more prone to them [3]. While chipkill can detect and correct more errors than ECC [4], it cannot do so for all memory errors and SDCs outside DRAM may not be detected at all, and such SDCs *are* routinely observed at scale [5]. The overall occurrence of bit flips is expected to increase as chip densities increase and feature sizes decrease.

While hardware vendors will address silent data corruption for the consumer and enterprise markets via more sophisticated detection and correction hardware logic, this will come at a price of increased memory overheads, latencies, and power. More importantly, it is not clear that these vendor solutions will be sufficient given the unprecedented scale and failure rates of future extreme-scale systems [6].

To address this SDC issue, we introduce FlipSphere, a software-based, generic memory protection library that increases application resilience by protecting data at the page level using an application transparent, tunable, and on-demand verification system with the following contributions:

- It introduces FlipSphere, which provides transparent protection against SDC for all applications without any program modifications.
- This is, to the best of our knowledge, the first published class of software-based resilience that harnesses hardware accelerators such as Xeon Phis to harden applications.

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

Further, this work evaluates the feasibility of full or shared co-execution for resilience on accelerators.
- We introduce two levels of protection and analyze their costs independently: Bit flip detection and bit flip detection plus automatic correction, thereby reducing the number of SDCs.
- FlipSphere provides a view into an application's memory access patterns as part of its functionality. In fact, using this feature, its operation is agnostic by tuning to the data access patterns of an application directly.
- It is extensible with new features, e.g., custom hashing algorithms or further strengthened ECC techniques.

II. Design

This paper introduces FlipSphere, a transparent, application-agnostic library that is capable of detecting and optionally correcting silent data corruption (SDC) in the memory of an executing process. FlipSphere works alongside traditional hardware ECC as an additional layer of defense against SDCs that exceed the limitations of hardware protection or it can independently provide protection on systems that lack hardware ECC altogether. Our SDC detection techniques are based on the ability to verify the correctness of memory residing in RAM that has not been accessed (relatively) recently. Specifically, we are most interested in ensuring that memory has not become perturbed by an SDC event before the application is allowed to read a region of memory. When FlipSphere guards memory accesses and verifies them before use, it is possible to ensure that memory read by a process has not been changed due to any external events, such as memory corruption.[1]

Fault Model: We assume that SDCs occur as random stochastic events uniformly distributed in DRAM due to radiation from space and fabrication miniaturization resulting in ever smaller differences between supply and threshold voltages. ECC may detect and correct some of these faults during memory scrubbing or when accessed, but not all as related work shows [2], [3], [4], [7], i.e., our method complements hardware scrubbing.

Error Model: FlipSphere currently protects an application's heap, BSS, and data sections but not that of the operating system. Although the FlipSphere methodology applies to all process sections, we protect neither the stack nor code (instructions) in the implementation. Not all faults result in errors, i.e., no error occurs when bit-flipped data is never referenced. We focus on HPC application that commonly operate on large data referenced in its entirety.

[1] An external event is defined as any occurrence that involves direct memory modification without FlipSphere's knowledge. For instance, FlipSphere must be made aware of DMA transfers before/after they occur, as they may bypass the CPU and thus FlipSphere.

A. Memory Access Tracking

The underlying tenant of FlipSphere is that an application's working-set of pages, which is the subset of virtual memory pages read or written to over a given period of time, is considerably smaller than the entire set of pages an application's data resides in. Further, FlipSphere assumes that SDC is likely to occur in memory that has not been accessed recently. We assume that the SDCs we are guarding against are more likely to occur in memory not recently accessed by the processor. In particular, FlipSphere is designed to protect memory in DRAM, as opposed to processor faults.

Next, we define two terms that relate to an application's data within FlipSphere: (1) *locked* memory, which has not been accessed recently, resides in RAM, and must be verified by FlipSphere before the application may read it, and (2) *unlocked* memory, which has been recently accessed and has undergone verification by FlipSphere to ensure its integrity before being read/used by the application. While the verification process will be explained later, the transition between *locked* and *unlocked* must be considered first. Let us assume that when an application is launched with 3 pages of data, all pages of its memory start in a *locked* state. Let us also assume an access pattern of P1, P3, P3, P2. Upon the first memory access to read memory in P1, FlipSphere will interrupt execution to verify P1 since it is *locked*. FlipSphere verifies that P1 is free from SDC (correct) via verification, marks it as *unlocked*, and returns control to the application. Next, P3 is accessed by the application, but since it is *locked*, the same aforementioned process occurs for P3 before control is returned to the application. When P3 is accessed a second time, no interruption occurs since it has already been marked as *unlocked* to indicate it was verified. Upon access of P2, the verification and transition to *unlocked* occurs again. At this point, all memory is marked as *unlocked*, and no further interruptions from FlipSphere occur since all pages are in this state.

FlipSphere strives to both minimize and ensure that the only pages allowed in an *unlocked* state are pages that have been recently accessed. A simple way to conceptualize this is to define a fixed-length least recently used (LRU) table of pages. Whenever a page is accessed, it will be added to the LRU table, and once the table is full, the oldest page entry in the table would be evicted to make room for the most recent page accessed. The eviction would require that FlipSphere transition the evicted page from an *unlocked* to a *locked* state, and simultaneously store some correctness metadata on the page so that it may later verify the page when it is accessed again in the future. As maintaining a LRU table and performing high frequency evictions/lookups would be prohibitively expensive in an HPC context, FlipSphere trades off the accuracy of an LRU table for the performance of a timer-based approach, wherein a *relock interval* is used to periodically transition all *unlocked* pages into a *locked* state.

Listing 1. "bench" example code

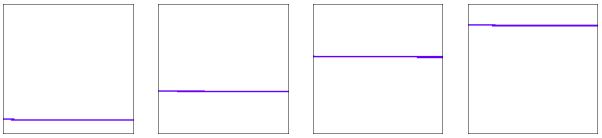
```
int* mem = malloc(sizeof(int) * outerloopmax * innerloopmax);
for(i=0..outerloopmax)
  for(BusyWorkLoop=0..100)
    for(j=0..innerloopmax)
      mem[i][j] += 1; //touch memory
```

Listing 2. "bench-rand" example code

```
int* mem = malloc(sizeof(int) * outerloopmax * innerloopmax);
for(i=0..outerloopmax)
  v = rand() % outerloopmax;
  for(BusyWorkLoop=0..100)
    for(j=0..innerloopmax)
      mem[v][j] += 1; //touch memory
```

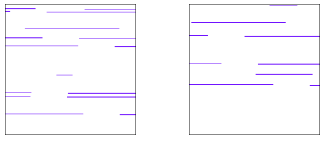
This approach allows for an unbounded number of pages to be accessed between each *relock interval* before all become *relocked*; however, in practice we tune the *relock interval* to a value that correlates to a target percent of memory being in the *unlocked* state. For example, in some applications, a *relock interval* of 0.1 seconds may correlate to no more than 10% of memory being *unlocked* on average.

Listing 1 demonstrates a sample application, henceforth "bench", that allocates a large array of memory and then proceeds to touch all memory in several iterations over the *i* loop. Using bench with FlipSphere, we can visualize the memory access patterns by utilizing FlipSphere's page access tracking functionality.



(a) f 10/83 (b) f 30/83 (c) f 50/83 (d) f 70/83

Figure 1. Memory access patterns of bench: *relock interval* 0.2 secs, frames 10,30,50,70 out of 83, blue=pages accessed in each frame, lower left corner=memory offset 0 with increasing addresses left to right and bottom to top.

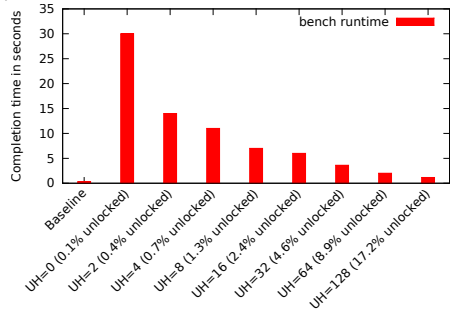


(a) f 12/93 (b) f 42/93

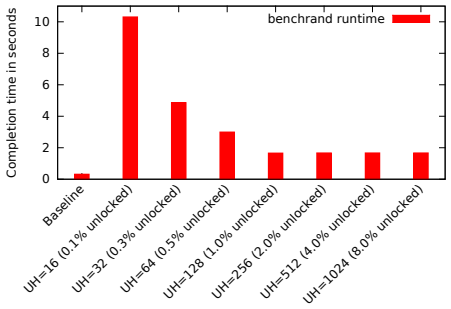
Figure 2. Memory access patterns of bench-rand: *relock interval*=0.2 sec.

In Fig. 1 we see four individual snapshots of the memory access pattern of bench during execution. To generate these graphs, we ran FlipSphere with bench while setting its *relock interval* to 0.2 seconds. At each interval, FlipSphere captured the status of every page's status. For instance, from Figures 1(a) to 1(b) we see that the blue *unlocked* pages have moved from lower addresses to higher addresses. In the example, an uninstrumented run of bench completed in 0.34 seconds, whereas the tracked execution in Fig. 1 was 100 times slower. Although the runtime overhead was severe, the amount of memory in the *unlocked* state stayed on average at approximately 0.1%.

To counter the high overhead of page-level tracking, we next introduce the optimization parameter *unlock ahead*. *unlock ahead* specifies a number of additional pages to unlock linearly ahead of the current page accessed. E.g., if *unlock ahead* is 3 while P10 is accessed, then pages P10, P11, P12, and P13 would all be unlocked at the same time. This reduces the number of interruptions an application will receive during execution, at the trade-off of a speculative memory access pattern. Tuning *unlock ahead* depends on the application's characteristics. For bench, memory is scanned from low addresses to high addresses without many jumps, so higher *unlock ahead* values will directly correlate to both an increase in performance and in the number of pages *unlocked* on average (see Fig. 3(a)). Listing 2/Fig. 2 show an application that reads 512KB chunks at random memory locations.



(a) bench: *relock interval*=0.2 secs



(b) benchrand: *relock interval*= 0.01 secs

Figure 3. Runtimes of benchmarks: UH=*unlock ahead* value, "% unlocked"=avg. % pages *unlocked* between each relock interval.

Fig. 3(b) demonstrates optimal performance (in terms of execution time) first at *unlock ahead*=128, since 128 * 4KB (VM Page Size) = 512KB. We also see that increasing the *unlock ahead* value beyond the working-set size of an application will **increase** the amount of memory *unlocked* without increasing performance. Specifically, we see that as we move from an *unlock ahead* of 128 to 256, 512, or 1024 that the only notable difference is a significant increase in the *unlocked* pages from 1% to 8% with no runtime effect. This effect underscores the importance of proper tuning since we do not want to speculatively unlock pages that will not actually be used by an application in the relatively near future.

B. Error Detection (Memory Verification)

By assuming the model for the previous section wherein memory accesses are efficiently tracked by FlipSphere, we next move to begin protecting memory from silent data corruption. In network communications and many file systems, cyclic redundancy check (CRC) codes are used to

Listing 3. Unlocking and Relocking (Detection only)

```
#define PAGE_SIZE 4096 /*Typical 4KB page size*/

int StoredCRCValues[TOTAL_PAGES]; /* 32bit int per page */
char* MemoryStart; /* Protected memory */

//Called upon page access. Transitions page from locked->unlocked.
void UnlockPage(int page) {
  void *address = MemoryStart + (page * PAGE_SIZE);
  if( GenerateCRC32(address, PAGE_SIZE) != StoredCRCValues[page] )
    ERROR(SDC_DETECTED, "Page %d had an SDC\n", page);
  MarkPageAsUnlocked(page);
}
//Called by a timer at relock-interval. Transitions ALL pages to locked.
void RelockAllPages() {
  for(int page = 0; page < TOTAL_PAGES; page++) {
    if(PageIsUnlocked(page)) {
      void *address = MemoryStart + (page * PAGE_SIZE);
      StoredCRCValues[page] = GenerateCRC32(address, PAGE_SIZE);
      MarkPageAsLocked(page);
} } }
```

guard against changes to raw data. CRC codes are one-way hashes that are particularly well suited to detect corruption of data as they are designed to be computationally cheap, fast to compute since they are used for verification instead of security. In FlipSphere we use the CRC32 hash, which requires only 32bits of storage per hash generated.

We can now extend the infrastructure provided by FlipSphere's page tracking concepts to define a means of memory verification that occurs during the transition between *unlocked* and *locked* states. First let us assume that for every page tracked by FlipSphere, we store additional metadata that contains both the *state* (*unlocked* or *locked*) and the *CRC*. If a page is in the *locked* state, then it is assumed to have a corresponding *CRC* saved in metadata that holds the CRC32 value of that page at the time when it previously transitioned from *unlocked* to *locked*. Further, when a page is accessed at the transition from *locked* to *unlocked*, we generate a new CRC32 of the page being accessed while FlipSphere interrupts the execution (Listing 3). The new CRC32 is compared against the previously stored CRC32. If they match then FlipSphere marks the page as *unlocked* and returns control to the application. However, if they do not match then the only conclusion is that the data was modified while it resided in RAM, which may indicate that an SDC has occurred. If we only perform CRC verification, FlipSphere may notify the user and/or process, and optionally terminate it immediately as to not perform further computations on invalid data. A rollback to a previously known good checkpoint may be desired, if checkpointing is available. However, we will later introduce automatic error correction provided by FlipSphere (in Section II-C).

Recall that after a page transitions to an *unlocked* state, there are no further interruptions from FlipSphere until it is returned to the *locked* state during the next relock interval. Since the CRC value stored by FlipSphere is immediately outdated once a page has been written to, FlipSphere must calculate and store a new CRC value every time the page transitions back to a *locked* state, as shown in Listing 3. This process occurs indefinitely throughout the execution of any application protected by FlipSphere.

Listing 4. Error Detection and Correction

```
#define PAGE_SIZE 4096 /*Typical 4KB page size*/

/* 72/64 Hamming stores 1 byte of ECC per 8 bytes of data */
#define ECC_GROUPS_PER_PAGE (PAGE_SIZE/8)
char StoredECCValues[TOTAL_PAGES][ECC_GROUPS_PER_PAGE];

int StoredCRCValues[TOTAL_PAGES]; /* 32bit int per page */
char* MemoryStart; /* Protected memory */

//Called upon page access. Transitions page from unlocked->locked.
void UnlockPage(int page) {
  void *address = MemoryStart + (page * PAGE_SIZE);
  if( GenerateCRC32(address, PAGE_SIZE) != StoredCRCValues[page] ) {
    PerformECCRecovery(address, StoredECCValues[page]);
    /* See if ECC was successful after attempted recovery */
    if( GenerateCRC32(address, PAGE_SIZE) != StoredCRCValues[page] )
      ERROR(SDC_DETECTED, "Page %d had an unrecoverable SDC\n", page);
    else
      printf("Page %d RECOVERED\n", page);
  }
  MarkPageAsUnlocked(page);
}
//Called by a timer per relock-interval. Transitions ALL pages to locked.
void RelockAllPages() {
  for(int page = 0; page < TOTAL_PAGES; page++) {
    if(PageIsUnlocked(page)) {
      void *address = MemoryStart + (page * PAGE_SIZE);
      StoredCRCValues[page] = GenerateCRC32(address, PAGE_SIZE);
      GenerateECCBits(address, StoredECCValues[page]);
      MarkPageAsLocked(page);
} } }
```

Table I
STORAGE OVERHEADS OF FLIPSPHERE ERROR DETECTION AND CORRECTION CODES

Algorithm	Overhead per 4KB page	Storage Overhead %
CRC32 Hash	4 bytes	0.10%
72/64 ECC	512 bytes	12.5%
Total Cost	**516 bytes**	**12.6%**

C. Error Correction

In the previous section, we referred to FlipSphere's ability to store a hash of pages that are under its protection. While the comparison of hash values cannot correct errors, we can still provide correction capabilities by computing and storing error correcting codes (ECC), such as hamming codes alongside our CRC hashes. For example, the 72/64 hamming code, frequently used in hardware, may be employed inside of FlipSphere to provide single error correction, double error detection (SECDED) capabilities at the expense of the additional storage required for the ECC codes. In fact, combining FlipSphere with hardware ECC can provide not only the ability to detect triple bit errors or greater (dependent on the capabilities of any particular ECC chip), but can also provide correction capabilities as the software-layered protection in FlipSphere may still retain viable error correcting codes once hardware protection has been exceeded.

Using FlipSphere extended with hashing plus ECC codes, it is possible to enjoy the protection and speed of hashing while limiting ECC code recalculation during unlocking only to times when a page has become corrupt during execution resulting in a mismatched hash. Listing 4 revises the pseudo-code for error detection (CRC) and correction (ECC). FlipSphere extends basic error detection by using 72/64 hamming codes for parity, which requires 1 byte (8bits) of storage per 8 bytes (64bits) of data, as shown in Table I.

D. Memory Layout

Internally, FlipSphere maintains several separate sections of memory, the largest of which is the memory of the protected application. At a high level, FlipSphere is capable of protecting any contiguous range of virtual memory, such as the application's heap or BSS data section exclusively. Beyond that, storage is kept for internal data, namely for *locked/unlocked* status, CRC, and ECC data of every managed page. The internal data is held in a separate range of virtual addresses distant from the application's data. Application data may be either (1) unprotected (outside of FlipSphere's range of protection), (2) protected but *unlocked* (if accessed recently), or (3) protected and *locked* (not recently accessed but with up-to-date CRC and ECC bits).

E. Regions of Memory Protected

Internally, FlipSphere maintains its own alternative protected heap space for the application and interposes memory allocation functions, such as `malloc`, `realloc`, and `memalign`. These interposed functions will allocate memory from FlipSphere's protected heap and provide addresses for the application. Later, these heap pages will always be either *locked* or *unlocked*.

Alternatively, for applications that allocate the bulk of their memory in a data or BSS section of the executable, FlipSphere protects those sections of data as well. When an application begins execution, all memory in FlipSphere's protected heap or data/BSS sections are *locked* by default. When an application allocates memory (i.e., calls `malloc`), the pages returned will become *unlocked* on future read/write memory accesses until locked by the library later on. Memory outside of the scope of FlipSphere's protected memory will by default not be protected or altered in any way.

F. Acceleration: Xeon Phi and SSE4.2

The heavy use of CRC hashing and software-based generation of ECC bits would normally prohibit the efficiency of any software error detection and correction mechanism. FlipSphere takes advantage of recent hardware developments by performing CRC generation on the host CPU by utilizing the new, performant CRC32 instruction[8] added in the SSE4.2 instruction set. Furthermore, one of FlipSphere's significant contributions is the use of accelerators with an asynchronous kernel written for the Intel Xeon Phi (Intel Many-Integrated Core) co-processors[9]. Each Phi co-processor is an Intel chipset with approximately 60 cores capable of over a teraflop of performance through a high degree of parallelism. As each co-processor is a dedicated PCI-x card, FlipSphere's custom kernel is responsible for data transfer of application memory via DMA between the host memory and the Phi's own on-board memory. From there, FlipSphere is able to generate new ECC bits (previously shown as a function in Listing 4) that are kept until the next relocking interval and the process is repeated. We will explore both full and partial Xeon Phi utilization for resilience such that a co-processor may optionally be partially co-executing an application and our resilience algorithm at the same time on a Phi.

G. Assumptions and Limitations

FlipSphere's protection extends only to memory and is not designed to protect against faults that occur in the CPU or other attached devices, including any attached Xeon Phi co-processors.

Since protection is provided for data stored in main memory, FlipSphere requires the capability to detect memory accesses. FlipSphere achieves this by altering process page tables and removing read/write page permissions in order to receive OS signals that indicate which memory addresses are being accessed upon a page fault. This requires a memory management unit (MMU) and the ability for applications to install a signal handler that detects access violations.

For simplicity, FlipSphere at present only protects memory that is dynamically allocated using previously mentioned functions such as `malloc` or static sections of data such as the BSS. While the benchmarks evaluated under FlipSphere handle BSS and `malloc` data, the approach generalizes and could be extended to protect all data regions (including code and initialized data).

As FlipSphere verifies page contents upon transitioning from the *locked* to the *unlocked* state, any SDCs that affect *unlocked* memory during the window in which they are not protected are vulnerable. For this reason it is important that the relocking timer fires frequently enough as to not needlessly leave more pages than necessary in an *unlocked* state when they are not being utilized. Specifically, it is desirable that the *relock interval* does not allow substantial amounts of memory to become *unlocked* when only a smaller working set of pages is needed for execution.

Any application that depends on DMA with devices such as network interconnects must ensure that buffers are in an *unlocked* state before DMA begins. This assumption is necessary since DMA avoids the MMU and thus FlipSphere is never notified of page accesses to buffers. Data written through DMA would appear as corruption to FlipSphere because changes made via DMA occur while the pages are in a *locked* state. As described in the next section, we ensure that MPI and other libraries depending on DMA safely work within this requirement via MPI library interpositioning. FlipSphere ensures that MPI safely works with *locked* pages used as pointers in MPI operations by tracking all outstanding MPI requests and any associated pointers.

III. Implementation

A. Memory Tracking Technique

To ensure protection of memory, FlipSphere has to receive a notification when a page is accessed, which is implemented via a `mprotect` system call to remove read and write access of protected pages. Removing these permissions ensures that a segmentation fault (`SIGSEGV`) violation is raised when the page is accessed. The library installs a signal handler for this `SIGSEGV` violation for notification.

Upon notification, FlipSphere uses an internal table to verify that the addressed page is under its protection. Then, as stated previously, verification is performed by comparing hash values. After verification, the page's read and write bits are restored, again using the `mprotect` call, and control is returned to the application.

FlipSphere's internal table stores the following information for each protected page.

- A status flag to indicate locked, unlocked, or permanently[2] unlocked pages;
- Storage for the page's last known good hash;
- (Optional) Storage for the page's software ECC bits.

Also, each locked page's hash and ECC storage is tunable to accommodate the size of whichever hashing algorithm is used.

B. Synchronized mprotect/TLB Flushing

In order for FlipSphere to track accesses to memory, it depends on the memory management unit (MMU) to trigger access violations whenever pages in the *locked* state are read or written. During the transition from *locked* to *unlocked*, which occurs after FlipSphere verifies an accessed page's hash, the `PROT_READ` and `PROT_WRITE` page permissions are added to the target page. In the x86_64 architecture, in which we evaluated FlipSphere, the process of adding additional access rights to a page does not involve flushing the TLB, which makes the cost of transitioning from `PROT_NONE` to read and write permissions relatively cheap. Unfortunately, the reverse process in which we return all pages' access rights to `PROT_NONE` during a relocking cycle is much more expensive and, in fact, causes a TLB flush for each call to `mprotect`.

To minimize the disruption caused by frequent TLB flushes, FlipSphere's unlock and relock algorithms are extended to perform as few `mprotect` calls as possible, by pre-computing which neighboring virtual pages will receive the same permissions in order to coalesce all identical page permissions into as few system calls as possible.

Further, if desired, FlipSphere may synchronize all protected processes on a node when the relocking timer is triggered. Instead of independently tracking the next relocking trigger per process, one server process simultaneously notifies all protected processes to temporarily suspend execution while pages are re-*locked* and `PROT_NONE` is applied to their page permissions. Synchronizing across all processes not only avoids costly TLB flushes during computation but also ensures that memory access performance remains as consistent as possible to reduce the system noise of TLB misses.

C. Hashing and ECC Implementations

FlipSphere currently supports the CRC32 hashing algorithm and the 72/64 hamming code (ECC) for both CPUs and the Xeon Phi accelerator. Additional hashing and/or ECC algorithms can be easily added to FlipSphere, e.g., utilizing external libraries such as libgcrypt or writing new kernels for the Xeon Phi. Our own 72/64 hamming code implementation is capable of single-error-correct-double-error-detect on each group of 64 bits in protected memory. Our kernel is capable of execution on either a host processor with SSE4.2 (we rewrote our algorithm to use instructions like the bit population counter in SSE4.2) or preferably on a Xeon Phi by substituting the population count op-code for a comparable one of the Phi's ISA.

We parallelize both the host and Phi code using OpenMP, substitute the normal parity computation with compact look-up tables, generate pipelined code by removing instruction branching, and finally unroll our computations to match the size of a virtual page on the target host. Note that we also implement verification/correction for the ECC codes to fix errors on-demand when a hash mismatch occurs.

D. Optimization: Background Relocking

Recall that FlipSphere can synchronously trigger all protected processes to relock their unprotected pages all at once to line up anticipated TLB flushes due to `mprotect`. Unfortunately, this leaves applications with a brief window of time in which all progress is stopped, including communication. This is FlipSphere's primary source of overhead to a protected application's completion time. Mitigating this, to allow applications to make forward progress in both computation and any outstanding communication, FlipSphere provides an optimized version of its relocking algorithm that operates in parallel to the application. By temporarily serializing the process of `mprotect`ing pages and optionally DMA copying memory to an accelerator, such as a Xeon Phi, FlipSphere can delegate the task of generating ECC bits to run in parallel to the application as soon the application's memory is safely copied via DMA.

[2]Pages considered permanently unlocked refer to any region of memory that is either undergoing a potential DMA transfer (MPI transmission, for instance) or a system call that may directly change application memory). Pages are no longer considered permanently unlocked as soon as the operation that placed them in this state completes, such as the completion of an MPI message receive to a buffer.

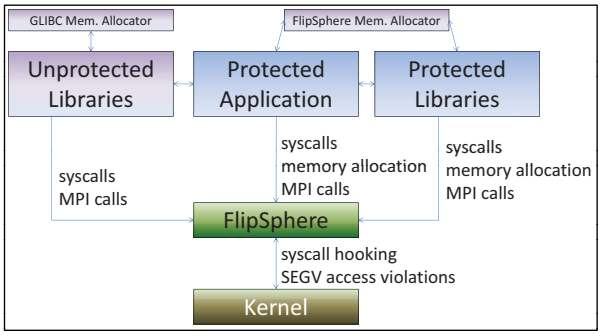


Figure 4. FlipSphere Component Interactions

Since accelerators operate in the background, the application continues with forward progress immediately.

E. Protected Memory: Heap and BSS

There are two potential types of memory that FlipSphere protects: the heap or BSS. First, heap data that is dynamically allocated via `malloc`, `realloc`, `free`, etc. is protected by interposing the standard memory allocation functions. Internally, a separate heap is created by FlipSphere and managed by the dlmalloc memory allocator to force the heap into a specific (known) memory range. Returning addresses that are within a predefined protected range of memory, FlipSphere's allocator effectively provides SDC protection for dynamically allocated memory. Since many dynamically linked libraries will make use of dynamic memory, our interposed functions will perform a back trace using the stack to identify if the caller should receive a pointer to the regular application heap or FlipSphere's protected heap. For example, MPI implementations will commonly dynamically allocate memory, but special care must be taken to never return pointers to a protected heap address due to the potential for DMA accesses between the MPI implementations and communication devices. Application/library `malloc` calls will receive protected memory.

Note that since MPI calls and system calls always carry the potential to include pointers to protected memory, FlipSphere monitors these calls from both protected and unprotected libraries as well as the application itself.

Figure 4 visualizes the interactions between the various software layers and components with FlipsSphere.

Alternatively, many applications opt to allocate large arrays of memory in their BSS section. For these types of applications, FlipSphere inspects the program during startup (using `LD_PRELOAD` and the `constructor` attribute we preempt even the `main()` function) and determines what range of virtual memory is allocated for the BSS section in order to directly apply page protection to this range. When FlipSphere is protecting the BSS, dynamic memory allocation interpositioning on `malloc` is disabled and thus no longer returns specially allocated *locked* memory to the application on heap requests. In this sense, FlipSphere mutually exclusively protects either the heap or BSS since we currently protect a single, consecutive memory range, which is an implementation decision made to facilitate the creation of a prototype library. It does not affect results as C/C++ codes tend to use heap while Fortran codes utilize the BSS, i.e., we did not encounter codes that extensively use both heap and BSS.

F. Client/Server Model of FlipSphere

As a single server may host multiple protected processes on it, FlipSphere has adopted a client/server model that allows it to provide services such as notification relock interval timers, multiplexing of Xeon Phi co-processors, and simplified logic abstracted away from running in the same address space as the protected application.

One of FlipSphere's primary goals is to be application agnostic. We accomplish this by bootstrapping the startup of a process to be protected by utilizing Linux's `LD_PRELOAD` environment to attach a FlipSphere shared library to the address space of a process during startup. This library proceeds to immediately intercept execution as soon as the process is created. A series of steps occur next:

- Config: FlipSphere reads configuration data from environment variables, including heap size, *unlock ahead* tuning, *relock interval* value, and other tuning parameters (thread/core counts, and whether to track memory, provide error detection, and/or utilize the Phi for error correction codes).
- Memory: FlipSphere allocates shared memory for the protected heap (if applicable) or unmaps the original BSS from the process's virtual mappings and then reallocates it as a shared region. It also creates shared memory for all internal data, such as page state, CRC values, and ECC bits.
- Fork: The original process forks, sharing only data segments allocated in the previous step. If a forked server already exists, the client instead attaches to the server, thus connecting to all of the client's shared memory segments.
- Server: The server tests communication with the client, and optionally "warms up" the Xeon Phi by allocating buffers which will later be filled with application memory prior to using the ECC calculation kernel.
- Client: The client installs signal handlers for `SIGSEGV` and a signal for *relock interval* timer events from the server. Special hooks are installed in all Linux signal manipulation functions to ensure the process cannot overwrite FlipSphere's signal handlers. Instead, they may be installed as chains by FlipSphere. Thereafter, FlipSphere code preloaded into the application will only execute in (1) the context of a signal handler for FlipSphere or `SIGSEGV`, (2) one of many hooked wrapper functions that ensure MPI and system calls have their pointers unlocked prior to execution, or (3) our alternative protected heap allocator, if applicable.

- Proceed: Full control is returned to the application, which begins to run `main()`.

One point of interest is that between the client (application) and server (FlipSphere), all virtual memory is shared via Linux shared memory. This removes the need for any redundancy in host memory or any memory copies, since both processes have access to the same physical memory, albeit in different virtual address spaces. By adopting this model, the server initiates communication via triggers and timers that are received as signals by the client. Utilizing inter-process communication (IPC), the server is able to control or multiplex the Xeon Phi for the client, which provides the added benefit that a protected application may still be able to offload computation on to a Phi if it desires since the server is an entirely separate process.

G. Handling User Pointers with System Calls

The use of the `SIGSEGV` handler allows FlipSphere to track an application's memory accesses at the page level during execution. Unfortunately, if an application or one of its libraries makes a system call with a pointer to userspace memory, the kernel will not invoke the userspace `SIGSEGV` handler when it is unable to dereference a pointer. E.g., a system call to `read` will directly place data into a userspace process's memory. Since the kernel is directly writing to a pointer of a process, a `SIGSEGV` will not be generated if the kernel is given a pointer to a region of memory that does not have write permission. For this reason, we must wrap relevant system calls and preemptively unlock any pointers to protected userspace memory that the kernel receives as parameters. To achieve this, FlipSphere includes hooks for syscalls via a wrapper that unlocks all pointers passed to the kernel prior to starting the actual system call's logic.

Not all system calls are interposed, and many do not even need to be since they lack userspace pointers. In choosing which system calls to support initially, we profiled many MPI applications and determined which system calls were needed for our MPI library and for the applications themselves to function as a proof-of-concept.

H. MPI Support and DMA

Any libraries (such as MPI) that depend on DMA to transmit data to hardware will operate outside the bounds of the MMU and thus will go undetected by FlipSphere. To ensure that any memory undergoing DMA accesses will not be perceived as corruption, FlipSphere interposes all calls to MPI functions using the MPI profiling layer. Any MPI function that takes as arguments a pointer to memory that will be read/written will be treated as permanently *unlocked* before control is transferred to MPI. Additionally, since non-blocking MPI sends and receives may leave a region of memory volatile to change for an indeterminate amount of time, FlipSphere tracks all outstanding MPI Request objects to ensure that any pending non-blocking MPI requests will be prevented from returning to the *locked* state until the MPI Request is completed. This specifically includes asynchronous send/receive operations, which FlipSphere identifies as outstanding until a matching `MPI_Test`, `MPI_Wait`, or related function is called for each request. Memory that may be modified by MPI and DMA operations will always remain in an *unlocked* state until the MPI operation that triggered it is finished. FlipSphere has support for all common MPI calls in C, C++, and Fortran.

IV. Experimental Results

To gauge FlipSphere's effectiveness, we performed experiments to demonstrate both range of coverage in memory and cost as application runtime overhead. These experiments were carried out on compute nodes of the Stampede cluster at TACC with each node consisting of 16 cores with Intel Xeon E5 E5-2680 processors and 32GB memory.

For our experiments involving Xeon Phi co-processors, we ran FlipSphere's kernels on Xeon Phi SE10P (Knights Corner) co-processors. Our results will cover performance metrics under utilization of an entire co-processor for resilience as well as partial resource sharing to allow co-execution of resilience algorithms and computation on the same accelerator.

As one of the contributions of FlipSphere is ECC generation in a Xeon Phi kernel, we first wrote our ECC algorithm on an x86_64 host CPU and evaluated the engine to measure the throughput of ECC generation. Next, we ported the optimized version of our algorithm to the Xeon Phi and evaluated it by varying the number of threads available. Table II shows the performance of both the host CPU and Phi. As our code is highly parallel, both configurations show peak performance when the maximum number of hardware threads is used. While raw throughput when all host CPUs are used does outperform a Xeon Phi at full capacity, it is critical to keep in mind that we may only dedicate a subset of the available computation resources to resilience while the majority of the resources will be for application progress. In FlipSphere, ECC generation occurs in tandem with regular computation, i.e., compute cores are used for application forward progress while others may be reserved and utilized for resilience. Crucially, if resources are not split between computation and resilience, then applications would be forced to wait for periodic resilience tasks to complete when in fact the tasks may be safely and efficiently split. Therefore, let us now look at the most important metric of Table II by noting that when 25% of the host threads are reserved for resilience, we achieve only 26% of the potential ECC generation (resilience) throughput. But we can nearly double the efficiency to 48% of optimal Phi performance

Table II
ECC GENERATION PERFORMANCE

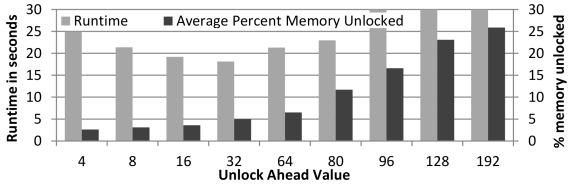
Number of Threads	Throughput	% of Optimum Throughput
1/16 CPU Thds.	1148 MB/s	7%
2/16 CPU Thds.	2161 MB/s	13%
4/16 CPU Thds.	4352 MB/s	26% **(25% host thds. used)**
8/16 CPU Thds.	8691 MB/s	51%
15/16 CPU Thds.	16016 MB/s	94%
16/16 CPU Thds.	**17035 MB/s**	100% **(Host Optimum)**
17/16 CPU Thds.	14800 MB/s	87%
61/244 Phi Thds.	4372 MB/s	48% **(25% Phi thds. used)**
122/244 Phi Thds.	7226 MB/s	76%
244/244 Phi Thds.	**9560 MB/s**	100% **(Phi Optimum)**
305/244 Phi Thds.	7754 MB/s	81%

when only 25% of Phi cores are reserve for resilience. The difference indicates that systems based on Xeon Phi are better suited for software-based resilience than traditional architectures.

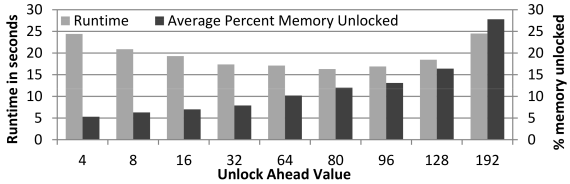
Although today's Intel Xeon Phi Knights Corner (KNC) co-processors are separate compute units that communicate through the PCI-express and maintain their own DRAM (separate from the host's DRAM), the next generation Xeon Phi (Knights Landing — KNL) will be available in two forms, as a co-processor and as the next generation host CPU. For FlipSphere's purposes, this change provides a key variable in the way we evaluate our performance. In KNC, which we evaluated FlipSphere on, all memory must incur a DMA between the host CPU and the co-processor via PCI-express. However, the next generation KNL simply will not incur the performance penalty as the host DRAM is directly attached to the Xeon Phi. While Xeon Phi's computational performance will likely improve in each generation, from FlipSphere's perspective we are most concerned with the fact that DMA overhead is not relevant and any clock speed performance gains or additional threads will only increase performance.

FlipSphere's Xeon Phi implementation of ECC generation was profiled earlier as shown in Table II. Our first result to report is an observed DMA transfer rate between the host CPU and Xeon Phi (KNC) of 6271 MB/s, which is lower than the kernel's ECC generation throughput of 9690 MB/s. Although the DMA transfer rate appears reasonable, we notice that the serial portion of FlipSphere requires a DMA copy from the application's data to the Xeon Phi before the application may resume progress. For instance, copying 512 MB of application data imposes a serial requirement of at least 0.08 seconds at every *relock interval*. For an application running for one hour with a *relock interval* of 0.1 seconds, we expect 72000 interrupts for DMA for a total costed time of 48 minutes, which would be unreasonable. As discussed, the DMA penalty will not apply to the KNL generation of Xeon Phi. Thus we will report results for KNC (DMA added), and KNL (no DMA, derived as an interpolation from the former) in the results section, although the pertinent result is the effectiveness of FlipSphere's Xeon Phi integration, not the cost of DMA, which will be phased out from Xeon Phi.

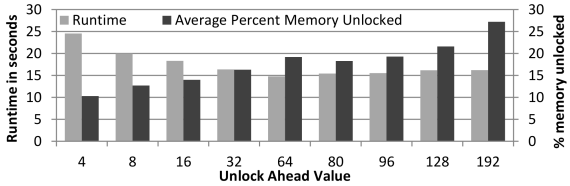
As FlipSphere protects applications from silent data corruption by employing a generalized technique of hashing pages to provide on-demand integrity verification, we chose to evaluate our library on two different applications from the NAS Parallel Benchmarks (NPB) suite. Our first experiment is NPB's FT (a discreet 3D fast Fourier Transform) benchmark with a customized input size 75x75x75 to extend execution across 10 iterations. The second experiment is NPB's CG (conjugate gradient solver) with a customized problem size of 150,000 and 10 iterations. Each application has different memory access patterns so that FlipSphere is evaluated in both an environment where memory is touched relatively infrequently per time unit (FT) and very frequently (CG). For reference, to consider how much memory will later be protected and transferred over DMA, CG processes used 475MB of memory, and FT 640MB.



(a) Relock interval 0.025 sec



(b) Relock interval 0.05 sec

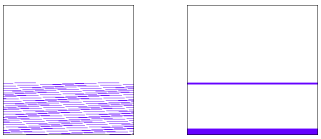


(c) Relock interval 0.10 sec

Figure 5. NAS FT CRC (Bit Flip Detection Only)

A. Error Detection (CRC) Only

We first analyze the performance of FlipSphere with error detection (CRC) enabled on the NPB-FT benchmark as shown in Fig. 5. In this particular experiment the host CPU is used for CRC generation, and Phi co-processor is not utilized. The three subfigures show multiple experiments with different *relock interval* values. Fig. 5(a) shows the results of varying the *unlock ahead* (UH) parameter while keeping a constant *relock interval* of 0.025 seconds. We observe a convex trend with a minimum runtime at UH=32 of 18 seconds (left y-axis) and approximately 5% memory unlocked (right y-axis) on average, or, inversely said, up to 95% of memory is protected against silent data corruption at this datapoint. *This is a substantial degree of protection.* For reference, the original benchmark completed in 10 seconds, which indicates that with RL=0.025 seconds and UH=32 we



(a) Strided (b) Linear

Figure 6. Two common, periodic memory access patterns of NPB-FT

Table III
NAS FT - CRC PLUS ECC - WITH DMA, 100% XEON THDS. (FULL LOAD ON KNIGHTS CORNER)

Row	Relock Interval	Unlock Ahead Value	Normalized Completion Time	Percent Memory Unlocked
1	0.05s	16	1.62x	17.8%
2	0.05s	32	1.46x	20.6%
3	0.05s	64	1.44x	23%
4	0.05s	128	1.51x	24.5%
5	0.1s	16	1.59x	18.3%
6	0.1s	32	1.5x	20.5%
7	0.1s	64	1.44x	23.1%
8	0.1s	128	1.45x	25.2%
9	0.15s	16	1.59x	22.8%
10	0.15s	32	1.51x	24.2%
11	0.15s	64	1.44x	26.7%
12	0.15s	128	1.52x	27.6%

observed a 70% runtime overhead, which is relatively high albeit with a high degree of protection.

As FlipSphere is designed to be tuned to an application, it is important to point out that the reported results show the affects of tuning. By observing the effects of varied input parameters we see that Fig. 5(c) shows no added runtime benefit (in terms of time to completion) when UH is increased beyond 32. In fact, as UH is increased, we only notice a steep decline in the amount of memory protected. Figures 5(a) and 5(b) both show sensitivity to *unlock ahead* and *relock interval*, which demonstrates the potential ranges a user would explore to find their desired trade-off of performance vs. resilience. Generally, decreasing the *relock interval* increases the resilience of an application and its time to completion, while the *unlock ahead* value is then subsequently used to further tune the application (decrease overhead by improving time to completion) to match both the working set size (per unit *relock interval*) and amount of contiguous memory read in stride per memory region/structure.

B. Error Detection and Correction

We now evaluate FlipSphere's detection and correction contributions in two configurations: (1) All Xeon Phi threads are used for resilience while the protected application runs on the host, which obligates DMA transfer between the host and Xeon Phi, and (2) we approximate a Xeon Phi running as the host processor (no DMA required) with 75% of threads reserved for the application while 25% are earmarked for resilience. Today's Knights Corner co-processors are used in the former and are an accurate representation of performance. An environment with a Knights Landing host processor is approximated by reserving only 25% of hardware threads for FlipSphere. The ratio of compute to resilience threads is variable, but we have chosen 25% from Table II. This configuration does not require DMA,

Table IV
NAS FT - CRC PLUS ECC - WITHOUT DMA, 25% XEON THDS. (APPROXIMATING LOAD SPLITTING)

Row	Relock Interval	Unlock Ahead Value	Normalized Completion Time	Percent Memory Unlocked
1	0.05s	16	1.56x	8.8%
2	0.05s	32	1.40x	10.8%
3	0.05s	64	1.58x	10.8%
4	0.05s	128	1.51x	15.4%
5	0.1s	16	1.47x	16.8%
6	0.1s	32	1.54x	16.8%
7	0.1s	64	1.43x	19.4%
8	0.1s	128	1.30x	23.3%
9	0.15s	16	1.54x	23.6%
10	0.15s	32	1.51x	24.6%
11	0.15s	64	1.37x	27.6%
12	0.15s	128	1.31x	29.5%

Table V
NAS CG - CRC PLUS ECC - WITH DMA, 100% XEON THDS. (FULL LOAD ON KNIGHTS CORNER)

Row	Relock Interval	Unlock Ahead Value	Normalized Completion Time	Percent Memory Unlocked
1	0.05s	16	1.80x	22.2%
2	0.05s	32	1.55x	31.3%
3	0.05s	64	1.35x	43.6%
4	0.05s	128	1.40x	50.4%
5	0.1s	16	1.69x	39.3%
6	0.1s	32	1.46x	53.4%
7	0.1s	64	1.33x	62%
8	0.1s	128	1.30x	66.3%

hence, performance is increased without the overhead of DMA. At the same time, the resilience throughput is decreased. Depending on the access patterns of applications, in some occasions the net difference results in no performance change in terms of protection and overhead. For FT, Rows 2 of Tables III and IV show similar completion time, for instance, but protection is halved in the latter. For CG, rows 1 and 2 of Tables V and VI show almost no difference in performance as the effect of moving the overheads from DMA to computation was balanced out.

For all results in Tables III to VI, we have highlighted what we deem to be the best performing configurations. Each row contains as tunable input parameters the *unlock ahead* value and the *relock interval* resulting in a normalized runtime (1.0x is an unmodified execution without resilience) and the percent of memory unlocked, on average. To find the degree of protection, we invert the percent of memory unlocked, i.e., lower unlocked is better. Although not shown, we have evaluated FlipSphere in many unreported configurations (to conserve space), but have reported the best performing values and their neighbors to provide perspective.

As shown in Fig. 6, FT predominantly experiences a strided access memory pattern and periodic scanning pattern. Compared to CG (which has rapid, linear scanning of all its address space), FT's performance with FlipSphere is highly dependent on a tuned *unlock ahead* value. While Table III is representative of performance with DMA, a similar result also appears in Table IV. We can see that in each set of 4 rows we only vary the *unlock ahead* value and always observe peak memory coverage between values 16-32 for *unlock ahead*. This is indicative of proper tuning to match the strides shown in Fig. 6. Beyond that range, memory coverage rapidly decreases for values such as 64-128. Although not shown, one can increase the mem-

Table VI
NAS CG - CRC PLUS ECC - WITHOUT DMA, 25% XEON THDS. (APPROXIMATING LOAD SPLITTING)

Row	Relock Interval	Unlock Ahead Value	Normalized Completion Time	Percent Memory Unlocked
1	0.05s	16	1.85x	21.9%
2	0.05s	32	1.52x	34.7%
3	0.05s	64	1.43x	42.1%
4	0.05s	128	1.21x	48%
5	0.1s	16	1.80x	41.8%
6	0.1s	32	1.43x	53.4%
7	0.1s	64	1.32x	62.7%
8	0.1s	128	1.38x	62.1%

ory coverage by lowering the *relock interval*, but this is prohibitively expensive as overheads quickly exceed 2-3x, which would then begin to favor double modular redundancy for bit flip detection alone (albeit without the corrective capabilities FlipSphere provides). For FT, we conclude that the highlighted row 2 is the best balanced configuration when both the completion time and memory protection are considered as it provides 90% memory coverage at a 40% overhead when Xeon Phi is the host CPU for both computation and resilience. Alternatively, it provides 80% coverage with a 46% overhead when traditional CPUs are used on the host for computation.

CG's performance is reported in Tables V and VI. While the tables feature different resilience configurations, the completion time and memory coverage is similar in CG's case since it is more memory bound than FT. While Table VI utilizes only 25% of the Phi's hardware threads for resilience, its loss of resilience throughput is balanced out by a lack of DMA transfer overheads (Knights Landing). In contrast to FT, the best reasonably achievable memory coverage of 88% is possible at a runtime overhead of 85% for both configurations of CG due to high frequency of memory scanning. CG illustrates the impact of high speed memory accesses on FlipSphere, still under protection at less than 2x overheads. Notice that we support error correction with overheads more performant than dual redundancy, which only provides error detection.

V. RELATED WORK

FlipSphere is based on the software of LIBSDC [10]. LIBSDC provides software-based page-level protection by tracking page accesses using the MMU and removing page permissions of a less recently used page every time a new page is accessed. LIBSDC may result in application slowdowns ranging from as little as 1x to as high as 20x due to a combination of its page locking mechanisms, dependence on the application tracing API `ptrace` and hashing methods used. FlipSphere differentiates itself from LIBSDC by providing a full software-based ECC implementation via hardware accelerators, protection of both application heap and BSS, non-blocking performant timer-based relocking instead of LRU, MPI and network device interception to provide protection for memory incurring DMA, and kernel function call tracking to remove dependence on `ptrace`. FlipSphere constitutes a low-overhead, feasible option for software-based SDC protection that uses hardware accelerators. Research involving GPGPUs to generate ECC codes of GPU memory has been investigated [11]. That work focused on generating and storing ECC codes for GPU memory that resides in GPU global memory only. In contrast, we generate software ECC for data the resides on the host CPU by using hardware accelerators.

One method to address silent data corruption is from the field of algorithmic fault tolerance where researchers have proposed methods to protect matrices from SDCs that corrupt elements within a dense matrix [12]. While these methods are effective for some matrix operations, fault tolerant algorithms are incredibly difficult to design for many arbitrary data structures or operations on that data [13]. Worse, this method does not provide comprehensive coverage to the entire application, which leaves anything outside of the algorithm vulnerable to SDCs. Distributed systems approaches and eventually consistent storage abstractions provide a software solution to harden limited state within protocols [14], [15] or checking code for specific data structures [16], but they do not generalize to large-scale numerical data (matrices) in HPC.

Another approach is to use software background scrubbing with ECC to periodically validate memory and correct errors if possible [17]. In contrast, FlipSphere provides on-demand page-level checking based on the application's data access patterns. In an HPC environment, software-based background scrubbing would likely consume considerable memory bandwidth and generate substantial application jitter [18].

Source-to-source transformation techniques [19] generate a redundant copy of all variables and code at the source level. Additional conditional checks verify agreement in the redundant variables after a set of redundant calculations is performed. If redundant variables do not agree, the application aborts. This technique is unable to handle pointers and requires frequent conditional jumps for consistency checking so that the efficiency of pipelining and speculative execution suffers. FlipSphere has lower memory overheads, supports pointers, and protects any region of memory at runtime.

SDC resilience in software. Error detection by duplicated instructions (EDDI) [20] duplicates instructions and memory and exploits super-scalar processor capabilities to redundantly execute all calculations and compares results between their redundant variable copies. Thus, memory is halved and register pressure is doubled. FlipSphere is platform-independent and does not require redundant execution. Extensions to EDDI have been proposed [21] that achieve better efficiency by assuming reliable caches and memory, but still require redundant registers and instructions. Their experiments showed an average normalized execution time of 1.41, but without protection for system memory. Compiled executables with fault tolerance were 2.40x larger than

the original codes. Control-flow checking tries to detect the effects of SDCs in applications [22] but does not protect against data corruption (SDCs).

VI. Conclusion

FlipSphere realizes a silent data corruption detection and correction library. It protects pages of memory with known good hashes per page coupled with software based ECC codes. Memory is verified on demand and FlipSphere confirms the integrity of each page upon access while fixing any potential errors using the precomputed ECC codes.

We showed that using Intel Xeon Phi co-processors to offload software-based resilience is feasible for adding a variable amount of memory resilience to an application using a generic approach, free of any type of algorithmic fault-tolerance requirements. Our research is the first to indicate that when a portion of computational resources is set aside for resilience, Intel Xeon Phi is superior relative to a traditional host CPU for software ECC resilience computation.

FlipSphere exceeds the standard of protection provided by ECC by ensuring that errors evading even hamming codes will always be detected, even if not fixed, by performing CRC32 verification after hamming code correction is attempted.

Our results indicated that FlipSphere's error detection and correction can achieve up to 90% coverage with a 40% runtime overhead for some classes of applications. Even highly memory bound applications still achieve up to 88% coverage at an overhead of 85%, which provides significant benefit over double redundancy (100% cost) or triple redundancy (200% cost) as FlipSphere further provides error correction, in contrast to detection only under dual redundancy. With costs between 40%-85% vs 200%, our results may indicate an opportunity to disable ECC for DRAM and rather run ECC-unprotected when FlipSphere provides protection for kernels (not just for ECC-detectable events but also extra protection against silent data corruption), particularly since turning off ECC may result in lower memory latency and power consumption; yet, in contrast to Li at al. [23], FlipSphere does not require algorithmic changes.

References

[1] C. Chen and M. Hsiao", "Error-correcting codes for semiconductor memory applications: A state-of-the-art review," *IBM Journal of Research and Development*, vol. 28, no. 2, march 1984.

[2] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild: a large-scale field study," in *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 2009.

[3] C. Di Martino, Z. Kalbarczyk, R. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of blue waters," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, June 2014.

[4] V. Sridharan and D. Liberty", "A study of dram failures in the field," in *Supercomputing*, Nov 2012.

[5] A. Geist, "How to kill a supercomputer: Dirty power, cosmic rays, and bad solder," *IEEE Spectrum*, Feb. 2016.

[6] K. Ferreira, R. Riesen, P. Bridges, D. Arnold, J. Stearley, J. H. L. III, R. Oldfield, K. Pedretti, and R. Brightwell, "Evaluating the viability of process replication reliability for exascale systems," in *Supercomputing*, Nov. 2011.

[7] A. Geist, "What is the monster in the closet?" Aug. 2011, invited Talk at Workshop on Architectures I: Exascale and Beyond: Gaps in Research, Gaps in our Thinking.

[8] "Fast crc computation for iscsi polynomial using crc32 instruction," White Paper, April 2011.

[9] "Intel xeon phi co-processor brief," White Paper, 2013.

[10] D. Fiala, K. Ferreira, F. Mueller, and C. Engelmann, "A tunable, software-based DRAM error detection and correction library for HPC," in *Workshop on Resiliency in High Performance Computing in Clusters, Clouds, and Grids*, 2011.

[11] N. Maruyama, A. Nukada, and S. Matsuoka, "Software-based ECC for GPUs," Jul. 2009.

[12] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. C-33, no. 6, june 1984.

[13] Z. Chen, "Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods," in *Symposium on Principles and Practice of Parallel Programming*, 2013.

[14] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini, "Practical hardening of crash-tolerant systems," in *USENIX Conference on Annual Technical Conference*, 2012.

[15] D. Behrens, C. Fetzer, F. P. Junqueira, and M. Serafini, "Towards transparent hardening of distributed systems," in *Workshop on Hot Topics in Dependable Systems*, 2013, pp. 4:1–4:6.

[16] C. Borchert, H. Schirmeier, and O. Spinczyk, "Generative software-based memory error detection and correction for operating system data structures," in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, June 2013.

[17] P. Shirvani, N. Saxena, and E. McCluskey, "Software-implemented edac protection against seus," *IEEE Transactions on Reliability*, vol. 49, no. 3, sep 2000.

[18] K. B. Ferreira, P. G. Bridges, and R. Brightwell, "Characterizing application sensitivity to OS interference using kernel-level noise injection," in *Supercomputing*, Nov. 2008.

[19] M. Rebaudengo, M. Reorda, M. Violante, and M. Torchiano, "A source-to-source compiler for generating dependable software," in *Workshop on Source Code Analysis and Manipulation*, 2001.

[20] N. Oh, P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, mar 2002.

[21] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *International Symposium on Code Generation and Optimization*, 2005.

[22] N. Oh, P. Shirvani, and E. McCluskey, "Control-flow checking by software signatures," *IEEE Transactions on Reliability*, vol. 51, no. 1, mar 2002.

[23] D. Li, Z. Chen, P. Wu, and J. S. Vetter, "Rethinking algorithm-based fault tolerance with a cooperative software-hardware approach," in *Supercomputing*, 2013.