SAND2012-1355C

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

David Fiala[1], Kurt B. Ferreira[2], Frank Mueller[1], Christian Engelmann[3], and Ron Brightwell[2]

[1] Department of Computer Science, North Carolina State University
{ dfiala | fmuelle }@ncsu.edu
[2] Scalable System Software, Sandia National Laboratories Albuquerque, NM 87123
{ kbferre | rbbrigh }@sandia.gov
[3] Oak Ridge National Laboratories engelmannc@ornl.gov

Abstract. Proposed exascale systems will present a number of considerable resiliency challenges. In particular, DRAM soft-errors, or bit-flips, are expected to greatly increase due to the increased memory density of these systems. Current hardware-based fault-tolerance methods will be unsuitable for addressing the expected soft error frequency rate. As a result, additional software will be needed to address this challenge. In this paper we introduce FlipSphere, a tunable, transparent silent data corruption detection and correction library for HPC applications.
FlipSphere provides comprehensive SDC protection for program memory by implementing on-demand page integrity verification. Experimental benchmarks show that using on-demand page verification with FlipSphere we can protect 80% to 50% of program memory with time overheads between 7% and 55%.

1 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 in memory) 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.

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 be successful at flipping bits in memory.

Current systems use memory with hardware-based ECC that is capable of correcting single bit error and detecting double bit errors [2] 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 DIMMs will incur correctable errors while 2%-4% will incur uncorrectable errors [11]), the frequency of higher bit errors is still a open research question. Nonetheless, the overall occurrence of bit flips is expected to increase as chip densities increase and feature sizes decrease.

While hardware vendors will address this issue of silent data corruption for the consumer and enterprise markets by adding more sophisticated detection and correction logic in hardware, this will come at a price of increased memory overheads, memory 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 [3].

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

- It provides transparent protection against SDC for all applications without the need for any program modifications.
- It operates agnostic to the data access patterns of an application.
- It is extensible. New features, such as custom hashing algorithms or software-based ECC, which can not only detect but also correct SDCs that evade hardware ECC can easily added.

2 Design

In this paper, we present FlipSphere, a application-transparent library that detects and corrects soft-errors in a program. 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. FlipSphere tracks memory accesses at the virtual memory page level and verifies that the contents of each accessed page have not unexpectedly been altered by corruption.

FlipSphere monitors all read and write requests that an application incurs during execution while simultaneously verifying these data accesses. This verification is done by comparing a current hash against a previously known good value for that page. If an unexpected hash mismatch occurs during execution, then FlipSphere will attempt to correct the flipped bits using the previously calculated and stored ECC bits for that page. After the ECC correcting algorithm completes, the page will be rehashed and verified again to ensure that the correction algorithm was successful. If the rehashed page still does not match, FlipSphere will terminate the process or roll back to a previous checkpoint. This ensures that the application does not continue to compute and report invalid results. Once a page's integrity has been successfully verified, the application is allowed to proceed with forward progress.

Once a memory page has been verified, it will be available for use without further interception by FlipSphere. A page in this state is referred to as *unlocked*. Likewise, all other pages that have not yet been verified by FlipSphere will be considered *locked*. For each additional *locked* memory access, FlipSphere will intercept the request and verify the *locked* memory before unlocking it and allowing the application to progress.

Page accesses(unlocking) can be thought of as such:

```
On page request (initial read or write):
  If page is locked:
    Perform hash of page
    Compare current hash with previously stored known good hash
    If any inconsistency found (hash mismatch):
      Attempt to correct the bit errors with software ECC
      Rehash the bad page again and compare
      If inconsistency still exists:
        Notify the presence of SDC and report location
        Terminate application / Rollback to previous checkpoint
    Mark page as unlocked
  Return control to application
```

As an application executes over time, it is inevitable that all needed pages within an application's address space will at some point become *unlocked*, which means that no further page-level error checking will occur. Therefore it is necessary for FlipSphere to occasionally put pages back in the *locked* state so that they may be protected from corruption.

Page locking involves the following steps:

```
On page relock event:
  For each unlocked page:
    Calculate new hash of entire page
    Generate ECC bits for data in page
    Storage hash and ECC bits in separate location
    Mark page as locked
  Return control to application
```

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 give the application addresses that later 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 at some later time.

As stated previously, to restore page-level protection FlipSphere must occasionally relock all of the protected pages. This process currently occurs based on a timer that triggers all FlipSphere processes to temporarily interrupt program execution and systematically inspect every page that was *unlocked* and return it to its *locked* state. During this transition from *unlocked* to *locked* a new hash of each page is calculated and optionally new software ECC bits are generated.

FlipSphere allows the user to tune the amount of time to wait between relocking all pages as the relock interval affects the execution time of an application. Specifying a relatively small interval leads to overheads due to more frequent hashing and ECC code generation. Additionally, in the current Linux-based implementation, the `mprotect` system call on multicore chips will result in the flushing of the translation lookaside buffer (TLB) for all cores, which can significantly reduces performance due to TLB misses as the application progresses. On the other hand, setting the relock interval to a relatively large value will reduce the effective protection that FlipSphere can provide.

Figure 1 shows a snapshot of an application's address space with this library. From the figure, we see some pages are unprotected and others reside in the protected heap as either *locked* or *unlocked*. The figure currently shows several *unlocked* pages, but as time goes on the set of *unlocked* pages will be reset at the next relock cycle. Note that as depicted, the FlipSphere internal data is always stored elsewhere in memory separate from the protected heap.

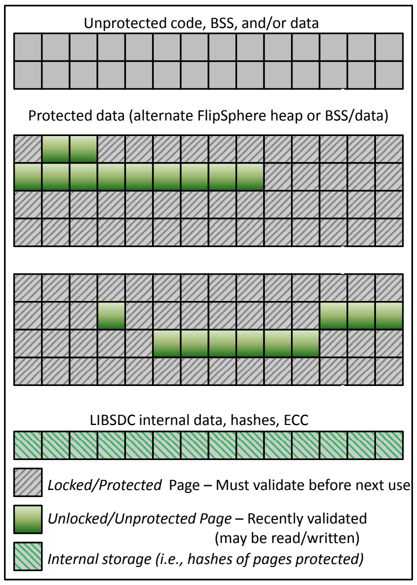

Unprotected code, BSS, and/or data
Protected data (alternate FlipSphere heap or BSS/data)
LIBSDC internal data, hashes, ECC
Locked/Protected Page – Must validate before next use
Unlocked/Unprotected Page – Recently validated (may be read/written)
Internal storage (i.e., hashes of pages protected)

Fig. 1. Memory Layout with FlipSphere

2.1 Extensions for Hashing and Error Correction

In the previous section of this paper we referred to FlipSphere's ability to store a hash of pages that are under its protection. This comparison of hash values is not capable for correcting errors. To provide correction capabilities, FlipSphere can additionally compute and store error correcting codes (ECC) such as hamming codes. 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. Table 1 provides a breakdown of the storage overheads associated with some common hashing algorithms and our ECC implementation.

With FlipSphere extended with hashing plus ECC codes it is possible to enjoy the protection and speed of hashing while limiting ECC code recalculation only to times when a page has become corrupt during execution resulting in a mismatched hash.

2.2 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. Since Flip-

Sphere uses page permission to track memory accesses it requires the use of a Memory Management Unit (MMU).

Any application that depends on DMA with devices such as network adapters must be handled specially since DMA may change memory without going through the MMU or notifying the OS of page accesses. FlipSphere ensures that MPI safely works with *locked* pages being used as pointers in MPI operations by tracking all outstanding MPI requests and any associated pointers.

Table 1. Comparison of Storage Overheads

Algorithm	Overhead per 4KB page	Storage Overhead %
CRC32 Hash	4 bytes	0.10%
MD5 Hash	16 bytes	0.39%
SHA1 Hash	20 bytes	0.49%
SHA256 Hash	32 bytes	0.78%
72/64 ECC	512 bytes	12.5%

3 Implementation

To ensure protection of memory, FlipSphere must be able to receive a notification when a page is accessed. To achieve this tracking, FlipSphere uses the `mprotect` system call to remove read and write access bits 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 unlocked pages;
– Storage for the page's last known good hash;
– (Optional) Storage for the page's software ECC bits.

3.1 Hashing and ECC Implementations

FlipSphere currently supports the CRC32 hashing algorithm and the 72/64 hamming code (ECC) for both CPUs and streaming accelerators such as a GPU. Additional hashing and/or ECC algorithms can be easily added to FlipSphere, for example, utilizing external libraries such as libgcrypt.

Our 72/64 hamming code implementation is capable of single-error-correct-double-error-detect on each group of 64 bits in protected memory. We also implement verification/correction for the ECC codes to fix errors on-demand if a hash mismatch occurs.

3.2 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. For this reason, we must wrap all system calls and preemptively unlock any pointers to protected userspace memory that the kernel receives as parameters. To achieve this, FlipSphere includes a counterpart kernel module that wraps system calls and directs the userspace FlipSphere library to unlock all pointers passed to the kernel prior to starting the actual system call's logic.

3.3 FlipSphere's Protected Memory: Protected Heap

There are two potential types of memory that FlipSphere protects: BSS or the heap. Since the BSS is the statically allocated portion of an application it is simple to protect. On the other hand, dynamic memory allocation requires special attention. By interposing the `malloc` family of functions, FlipSphere can be tuned to specify which additional libraries will receive pointers to protected memory in an alternate, protected heap. For instance, using a stack backtrace whenever `malloc` is called, FlipSphere chooses whether to allocate space in the protected heap, or whether to give back a normal, unprotected allocation via LIBC. This is useful to exclude libraries such as MPI from internally allocating data that may later be used in DMA operations. Figure 2 demonstrates the separation of heaps for libraries that receive protected memory or unprotected libraries.

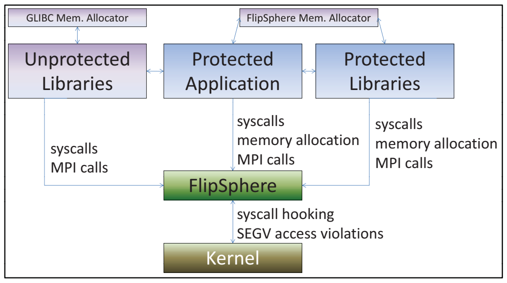


Fig. 2. FlipSphere Component Interactions

3.4 Optimization: Synchronized mprotect/TLB Flushing

In order for FlipSphere to track access 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 synchronizes all processes on a node when the relocking timer is triggered. Instead of independently tracking the next relocking trigger per process, one master 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.

3.5 Optimization: Background Relocking

As previously described, FlipSphere synchronously triggers all processes on the same node to relock their unprotected pages all at once. Unfortunately, this leaves applications with a brief window of time in which all progress is stopped, including communication. 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 protected pages, FlipSphere can delegate the task of rehashing pages that just got locked to a background thread or streaming accelerator. Since this thread operates in the background while the application continues with forward progress, both the application and the background relocker will compete to be the first to either access or lock/rehash pages. Pages accessed first by the application will not be hashed and are considered “skipped”. Simultaneous accesses result in a “collision” and are still hashed before being accessed.

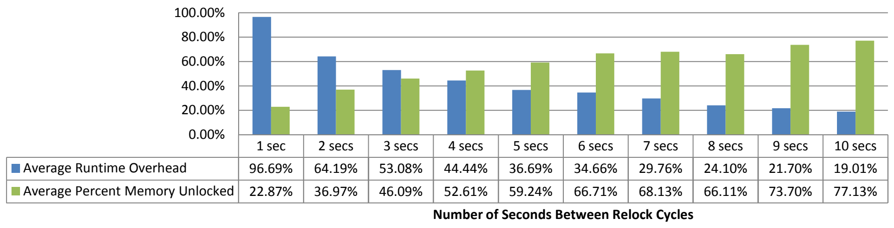
3.6 Optimization: Accelerators

Due to the nature of ECC generation, hashing algorithms, and the structure of FlipSphere's internal storage, there is ample opportunity for parallelism and the use of streaming accelerators. We include in FlipSphere an implementation for both CPU hashing/ECC generation as well as support for accelerators such as AMD Fusion accelerated processing units (APUs) or NVIDIA's GPUs. Additionally, FlipSphere exploits hardware features such as the SSE4.2 CRC32 instruction which provides accelerated CRC32 generation at demonstrated rates of over 21 gigabytes per second per CPU thread. [1]

4 Experimental Framework

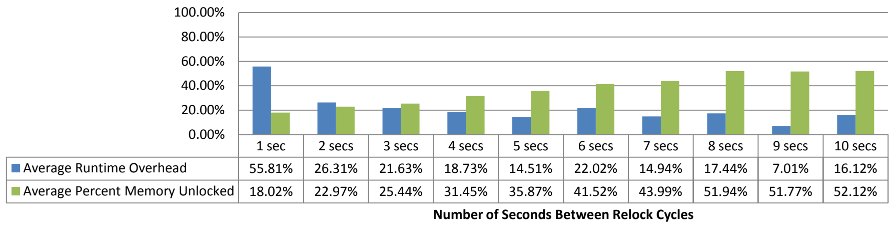
To gauge FlipSphere's effectiveness we performed experiments that demonstrate both its range of coverage in memory and cost in terms of application runtime overhead. These experiments were carried out on compute nodes of an HPC cluster with each consisting of 2-way SMPs with AMD Opteron 6128 and 32GB memory. As we are predominantly interested in the effects of application performance on a per-node basis, we chose to run experiments that saturated a single node at a time. Scaling across multiple nodes that are equally saturated should incur the same performance characteristics with FlipSphere as single-node performance.

As our experimental cluster is only equipped with NVIDIA GPUs, we simulate the expected time it would take for a shared-die APU accelerators to execute page hashing and ECC generation. To obtain a realistic expected cost for hashing and ECC generation, we ran our own custom implementations of NVIDIA CUDA kernels for hashing (CRC32) and ECC (72/64 hamming codes), but only count the time spent in the computation kernel while excluding DMA overhead. Since we expect that shared-die APUs will incur the same DRAM access latencies that NVIDIA GPUs do when they access their own global memory. Similarly, as our CPUs lack the new SSE4.2 instruction set, we simulate the CPU time needed for CRC32 generation using performance metrics provided by Intel [1] on the CRC32 instruction.



	1 sec	2 secs	3 secs	4 secs	5 secs	6 secs	7 secs	8 secs	9 secs	10 secs
🟦 Average Runtime Overhead	96.69%	64.19%	53.08%	44.44%	36.69%	34.66%	29.76%	24.10%	21.70%	19.01%
🟩 Average Percent Memory Unlocked	22.87%	36.97%	46.09%	52.61%	59.24%	66.71%	68.13%	66.11%	73.70%	77.13%

(a) NAS Parallel Benchmarks - FT



	1 sec	2 secs	3 secs	4 secs	5 secs	6 secs	7 secs	8 secs	9 secs	10 secs
🟦 Average Runtime Overhead	55.81%	26.31%	21.63%	18.73%	14.51%	22.02%	14.94%	17.44%	7.01%	16.12%
🟩 Average Percent Memory Unlocked	18.02%	22.97%	25.44%	31.45%	35.87%	41.52%	43.99%	51.94%	51.77%	52.12%

(b) NAS Parallel Benchmarks - LU

Fig. 3. Reported runtime overheads of FlipSphere with average percent of memory unlocked at any given time over varied relock intervals

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, each with varying memory access patterns. Our first experiment is NPB's FT (a discreet 3D fast Fourier Transform) benchmark with class C input size. The second experiment is NPB's LU (Lower-Upper Gauss-Seidel solver) with a customized problem size of 300 and 12 iterations maximum. Both experiments were performed with 8 MPI processes run concurrently on the same node. For our baseline experiments that lack FlipSphere protection, FT class C completes 20 time steps in 108 seconds on average while our custom input to LU completes 12 time steps in 180 seconds. This important difference in iteration time will later allow us to see how the speed at which memory is accessed (i.e., number of seconds needed per iteration) will affect the protection and efficiency of FlipSphere in our results.

We are predominantly interested in the effects of application performance on a per-node basis, i.e., we chose to run experiments that saturated a single node at a time. Scaling across multiple nodes that are equally saturated does not add additional insight since it incurs the same performance characteristics with FlipSphere as single-node performance.

5 Results

FlipSphere's CUDA implementations of CRC32 hashing and 72/64 ECC code generation were evaluated on a NVIDIA GTX 480. Timing only the kernel execution time, we observed that once properly tuned we could achieve a maximum hashing rate of 44 milliseconds/GB and 36 milliseconds/GB for ECC generation.

Combined, our simulated cost using an APU for hash and ECC generation that occurs at each relocking interval within experiments was 80 milliseconds/GB.

First, we analyze the results of the NPB FT benchmark. At this problem size, FT allocated a total of 844MB of memory for computation. Figure 3(a) indicates that we can provide a wide range of coverage with only 23% of memory left unlocked and thus unprotected on average for a one second FlipSphere relock interval. However, this high degree of coverage also cost nearly 100% in terms of time overhead due to the high frequency at which pages must be rehashed and returned to a *locked* state. Across the spectrum of relock intervals for FT, we see that, by the time we relock pages with an interval of 4 seconds, our memory left unprotected grows to over 50%. Finally, with a 10 second relock interval memory coverage has dropped to merely 23%, yet with time overhead of just 19%.

What can be observed here is that with such a high rate of accessing a breadth of the application's pages, FlipSphere's performance quickly suffers. Although it is still possible to protect an application, its memory tend to be touched quickly, and additional computational overhead will be incurred to verify all of these accesses.

Next, we investigate the costs of the NPB LU benchmark which has been modified to use a custom input. Unlike our previous experiment with FT, we have change the input size of LU to use 566MB of memory per process and to complete relatively fewer time steps in a longer interval. By increasing the time required to complete a each pass of a time step, the memory access pattern of LU is effectively being slowed down, which generates a smaller working set of memory relative to its entire resident set.

Figure 3(b) shows that LU with FlipSphere is capable of protecting the majority of memory even with relocking intervals that approach 10 seconds. Additionally, since LU is operating with a working set size that FlipSphere is able to hash in relatively little time. Overheads of 56% to as low as 7% are considerably cheaper and palatable than those seen with applications experiencing high rates of touching all memory.

These results indicate that for applications that maintain a large resident set of memory but temporarily use a smaller working set for computation, FlipSphere can provide effective SDC coverage to the majority of the non-working memory set.

6 Related Work

FlipSphere is a redesign of LIBSDC [5]. LIBSDC provided 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 was accessed. LIBSDC reported application slowdowns of over 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, protection of both application heap and BSS, timer-based relocking instead of LRU, non-blocking performant background relocking, hugepages support, and a lightweight kernel module to remove dependence on `ptrace`.

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 [6]. While these methods are effective for many matrix operations, such as multiplication, it is not clear that this form of fault tolerance can protect all types of possible matrix operations even if we disregard the fact that matrices are only one of numerous important structures. Additionally, although promising in many regards, fault tolerant algorithms are incredibly difficult and sometimes impossible to design for many arbitrary data structures or operations on that data. Worse, this type of protection does not provide comprehensive coverage of the entire application, which leaves anything outside of the algorithm, such as other data and instructions, entirely vulnerable to SDCs.

Similar to FlipSphere, another approach uses software-implemented error detection and correction using background scrubbing combined with software ECC to periodically validate all memory and correct errors if possible [12]. While this approach and FlipSphere are both entirely transparent to the application, FlipSphere differentiates itself by providing on-demand page-level checking based on the application's data access patterns. In a HPC environment, software-based background scrubbing would likely consume too much of the already limited memory bandwidth and generate substantial application jitter [4] during execution.

While the techniques described thus far in this section do not require changes to the application or executable, the techniques described in the remainder of the section require either modifications of the executable through source-to-source transformations or modifications to the executable with duplicate instructions or control flow verification.

Source-to-source transformation techniques [9] have been investigated that generate a redundant copy of all variables and code at the source level. Throughout the transformed source code are additional conditional checks that verify agreement in the redundant variables after each set of redundant calculations are performed. If at any point throughout the execution redundant variables do not agree then the application aborts. Unfortunately, however, this technique is unable to handle pointers, only supports basic data-types and arrays, and doubles the required memory. SDCs that occur in the instruction memory may not be detected, thus causing unpredictable results. Due to a high number of conditional jumps used for consistency checking, the efficiency of pipelining and speculative execution suffers. FlipSphere differentiates itself from this work by not requiring source modifications, lowering the memory requirement overheads substantially, supporting any type of code (pointers, data-types, etc are irrelevant to FlipSphere), and can be instructed to protect any region of memory at run-time.

Duplication instructions is another proposed technique to increase SDC resilience in software. Error detection by duplicated instructions (EDDI) [7] duplicates instructions and memory in the compiled form of an application in a manner similar to the source-to-source transformations, but achieves more support for programming constructs at the cost of platform dependence. Unlike the source-to-source transformations, EDDI compiles applications to binary form, redundantly executes all calculations, ensures separation between calculations by using differing memory addresses and differing registers, and attempts to order instructions to exploit super-scalar processor capabilities. During execution, the results of calculations are compared between their redundant variable copies, but as a result, available memory is halved and register pressure is doubled. FlipSphere differentiates itself from this work by being platform-independent, not requiring redundant execution or program modifications, and protecting instruction memory without the need for complex control-flow checks.

Extensions to EDDI have been proposed [10] 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. The similarity to EDDI may indicate that even without protecting memory there is a substantial overhead due to register pressure, additional instructions, and highly frequent conditionals that come with duplicating instructions and registers. This work also showed that compiled executables with the added fault tolerance were 2.40x larger than the original unaltered executables.

Control-flow checking is another area of research that attempts to detect the effects of SDCs in applications [8]. Unfortunately control-flow integrity verification does not necessarily protect against SDCs that only alter data without affecting the execution path of an application.

7 Conclusion

In this paper we have presented an implemention of our silent data corruption detection and correction library, FlipSphere. FlipSphere operates by protecting pages of memory with known good hashes of each page coupled with software based ECC codes. Memory is verified in an on-demand fashion and FlipSphere confirms the integrity of each page upon access while fixing any potential errors using the precomputed ECC codes. FlipSphere is transparent to the applications that it protects and is capable of being tuned to allow an operator the choice of which application memory to protect, the desired degree of memory coverage vs time overhead, and the choice of alternative hashing or ECC algorithms. Our implementation of FlipSphere is capable of exploiting both accelerated processing units (APUs), GPUs, and hardware features such as SSE4.2 to provide low overhead options for hashing and ECC generation.

Our results show that FlipSphere is capable of providing low overhead protection to the vast majority of memory for applications that exhibit temporal locality within their working-set of memory. This work shows great potential for SDC protection: experimental results demonstrate time overheads of only 7% while still protecting 50% of memory for the NPB's LU benchmark. Tuned differently, for LU we can also see protection of up to 82% of memory with a time overheads of 55%.

With its wide variety of tunable options, FlipSphere constitutes a low-overhead feasible option for software-based SDC protection.

References

1. Fast crc computation for iscsi polynomial using crc32 instruction. White Paper (April 2011), `download.intel.com/design/intarch/papers/323405.pdf`
2. Chen, C.L., Hsiao, M.Y.: Error-correcting codes for semiconductor memory applications: A state-of-the-art review. IBM Journal of Research and Development 28(2), 124 –134 (march 1984)
3. Ferreira, K., Riesen, R., Bridges, P., Arnold, D., Stearley, J., III, J.H.L., Oldfield, R., Pedretti, K., Brightwell, R.: Evaluating the viability of process replication reliability for exascale systems. In: ACM/IEEE Conference on Supercomputing (SC'11) (Nov 2011)
4. Ferreira, K.B., Bridges, P.G., Brightwell, R.: Characterizing application sensitivity to OS interference using kernel-level noise injection. In: ACM/IEEE Conference on Supercomputing (SC'08). p. 19 (Nov 2008)
5. Fiala, D., Ferreira, K., Mueller, F., 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. Springer Verlag, Berlin, Germany, Bordeaux, France (Aug 29 - Sep 2, 2011)
6. Huang, K.H., Abraham, J.: Algorithm-based fault tolerance for matrix operations. Computers, IEEE Transactions on C-33(6), 518 –528 (june 1984)
7. Oh, N., Shirvani, P., McCluskey, E.J.: Error detection by duplicated instructions in super-scalar processors. Reliability, IEEE Transactions on 51(1), 63 –75 (mar 2002)
8. Oh, N., Shirvani, P., McCluskey, E.: Control-flow checking by software signatures. Reliability, IEEE Transactions on 51(1), 111 –122 (mar 2002)
9. Rebaudengo, M., Reorda, M., Violante, M., Torchiano, M.: A source-to-source compiler for generating dependable software. In: Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE International Workshop on. pp. 33 –42 (2001)
10. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I.: Swift: Software implemented fault tolerance. In: Proceedings of the international symposium on Code generation and optimization. pp. 243–254. CGO '05, IEEE Computer Society, Washington, DC, USA (2005), `http://dx.doi.org/10.1109/CGO.2005.34`
11. Schroeder, B., Pinheiro, E., Weber, W.D.: Dram errors in the wild: a large-scale field study. In: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems. pp. 193–204. SIGMETRICS '09, ACM, New York, NY, USA (2009), `http://doi.acm.org/10.1145/1555349.1555372`
12. Shirvani, P., Saxena, N., McCluskey, E.: Software-implemented edac protection against seus. Reliability, IEEE Transactions on 49(3), 273 –284 (sep 2000)