A Tunable, Software-Based DRAM Error Detection and Correction Library for HPC

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

[1] Department of Computer Science, North Carolina State University
`{dfiala,fmuelle}@ncsu.edu`
[2] Scalable System Software, Sandia National Laboratories, Albuquerque, NM 87123
`kbferre@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 LIBSDC, a tunable, transparent silent data corruption detection and correction library for HPC applications. LIBSDC provides comprehensive SDC protection for program memory by implementing on-demand page integrity verification. Experimental benchmarks with Mantevo HPCCG show that once tuned, LIBSDC is able to achieve SDC protection with 50% overhead of resources, less than the 100% needed for double modular redundancy.

1 Introduction

With the increased density of 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 in to computing centers as our computational demands expand are posing a strong challenge to the HPC community. Of particular interest are soft errors in memory that manifest themselves as silent data corruption (SDC).

⋆ Sandia National Laboratories is a multiprogram 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.

⋆⋆ Research sponsored in part by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory (ORNL), managed by UT-Battelle, LLC for the U.S. Department of Energy under Contract No. De-AC05-00OR22725. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

SDCs are of great importance due to their ability to render invalid results in scientific applications.

Silent data corruption can occur in many components of a computer 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 the densities continue to grow, any event that upsets chips (i.e., radiation) is more likely to both interact with and be successful at flipping bits in memory.

Currently, servers that use memory with hardware-based ECC are capable of correcting single bit error and detecting double bit errors [1], but errors that result in three or more bit flips will produce undefined results including silent data corruption which may produce invalid results without warning. Today, research has been performed on the frequency and occurrence of single and double bit errors [9], but data on the frequency of triple bit errors remains inconclusive even though up to 8% of DIMMs will incur correctable errors while 2%-4% will incur uncorrectable errors. Nonetheless, the overall occurrence of bit flips is expected to increase as chip densities increase and computing centers move to millions of cores.

To combat this growing problem, new methods to both detect and correct faults that result in data corruption are essential. Specifically, it is critical to develop a fault resilient framework that provides for SDC detection and continuous execution in the face of faults. As applications increase in run time and scale out, it is no longer feasible to rely on traditional checkpoint-restart solutions to protect an application. Even with the bottlenecks that are checkpoint/restart I/O aside, we can not guarantee that an execution will be able to fully execute fault-free without interrupt due to a low average time between failures. Following this thought, we may not be able to reliably verify application results by simply running it twice if we are prone to a very high probability that a fault will render the results of both runs incorrect.

One method to address silent data corruption is in the field of algorithmic fault tolerance where researchers have proposed methods to protect matrices from SDCs that corrupt elements within a matrix [3]. While it is possible for this work to protect some matrix operations such as multiplication, this form of fault tolerance may not be able to protect all types of possible matrix operations even if we disregard the fact that matrices are only one of numerous important types of structures. Although promising in some regards, fault tolerant algorithms can be incredibly difficult or simply impossible to design for any arbitrary data structure or operation on 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.

For these reasons, there is a dire need to develop generic fault tolerance options that provide wide coverage to an application and its data while remaining agnostic to the actual algorithms that applications utilize.

This paper outlines a generic memory protection library that increases the resilience of all applications that it guards by protecting data at the page level using a transparent, tunable on-demand verification system. The library presented within provides the following contributions:

– Provides transparent protection against SDC for all applications without the need for any program modifications.
– Our solution is tunable to best match the data access patterns of an application.
– Extensibility within the library provides for easy addition of new features such as adding software-based ECC which can not only detect, but also correct SDC that evades hardware ECC.

2 Design

In this paper we present LIBSDC, a transparent library that is capable of detecting and optionally correcting soft-errors in system memory that cause corruption in program data during execution. LIBSDC protects against SDCs by tracking memory accesses at the virtual memory page level and verifies that the contents of each accessed page have not unexpectedly been altered.

To ensure memory has not become corrupted, LIBSDC is responsible for monitoring all read and write requests that an application incurs during execution while simultaneously verifying these data accesses. Each memory access is henceforth assumed to be at the granularity of an entire page of virtual memory instead of individual bytes. At a high level, each memory access that an application makes will be intercepted by LIBSDC and the contents of the page in which the memory address resides are verified against a previously known-good hash of that page. If during execution an unexpected hash mismatch occurs between the page and its last known value, then LIBSDC will terminate the process or roll back to a previous checkpoint if available to ensure that the application does not continue to compute and report invalid results. After a page's integrity has been successfully verified, then application is allowed to proceed with the memory access and continue making forward progress.

Once a memory access completes verification, the entire page in which the access resides will become available for use without further interception by LIBSDC. A page in this state will be referred to as *unlocked*. Likewise, all other pages that have not yet been verified by LIBSDC will be considered *locked*. For each additional *locked* memory access that occurs, LIBSDC 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:
      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 LIBSDC to occasionally put pages back in a *locked* state when they are no longer being used so that they may be protected from SDCs while resident in memory.

Page locking is shown as follows:

```
On page lock:
  Calculate new hash of entire page
  Storage hash in separate location
  Mark page as locked
  Return control to application
```

Managing *locked* and *unlocked* pages internally requires LIBSDC to hook memory allocation functions such as `malloc`, `realloc`, and `memalign` to learn of new memory addresses that should receive protection. When a new memory range has been allocated for an application, LIBSDC automatically locks all pages in the range of the new memory so that all future accesses to that memory are within the scope of protection that LIBSDC provides.

As the amount of allocated memory per application as well as the working-set of pages required varies, LIBSDC allows the user to tune the maximum number of pages to allow in the *unlocked* state. This tunable parameter, known as *max-unlocked*, is set prior to invoking an application and permanently defines the maximum number of pages to allow *unlocked* at any given time during execution. When the *max-unlocked* limit of *unlocked* pages is reached, any further accesses to pages in the *locked* state will require LIBSDC to lock some other *unlocked* page to accommodate for the new page of memory.

Tuning the *max-unlocked* parameter requires consideration as its value is directly related to both application performance as well as the effectiveness of SDC protection. Providing a relatively low *max-unlocked* value will force LIBSDC to more frequently lock and unlock pages resulting in unnecessary verifications. In this case, the overhead of intercepting page accesses combined with frequent rehashing will quickly diminish application performance. The effect of a *max-unlocked* value much less than the application's work-set of pages will result in a reaction comparable to thrashing. On the other hand, if the *max-unlocked* value is set too high, (i.e. a value much greater than an application's working-set) then the maximum level of SDC protection afforded by LIBSDC might not be attained. Too high a *max-unlocked* value will affect pages that remain *unlocked* for long periods of time without use while leaving them vulnerable to SDCs as their contents are only protected once switched back to the *locked* state. For these reasons it is important to tune applications using LIBSDC with an reasonable *max-unlocked* value that adequately expresses the level of protection vs. overhead desired.

2.1 Extensions for Error Correction

Through the design section of this paper we have referred to LIBSDC storing a hash of pages that are under its protection. When a page is hashed, the hash may be compared against a future hash taken on the same page to determine if any changes have occurred, but this information alone is not suitable for correcting errors that a hash may detect. To provide additional SDC correction capabilities on top of the detection mechanisms, it is possible to additionally compute and store error correcting codes (ECC) such as hamming codes that may be used to fix bit flips in memory. For example, 72/64 hamming codes which are frequently used in hardware may be employed inside of LIBSDC to provide single error correct, double error detect (SECDEC) protection at the expense of the additional storage required for the ECC codes. Combining LIBSDC with hardware-ECC can provide not only the ability to detect triple bit errors or greater, but can also provide correction capabilities as the software-layered protection in LIBSDC may still retain viable error correcting codes. If LIBSDC is extended with hashing plus ECC codes then it is possible to enjoy the protection and speed of hashing while limiting ECC code recalculation only to times when a page has been modified during execution resulting in a changed hash.

2.2 Assumptions and Limitations

LIBSDC's protection extends only to memory and is not designed to protect against faults that occur in the CPU or other attached devices. Since protection is provided for data stored in main memory, LIBSDC requires the capability to detect memory accesses. LIBSDC achieves this by altering process page tables and removes read/write page permissions in order to receive OS signals that indicate which memory addresses are being accessed upon a page fault.

For simplicity, our prototype of LIBSDC at present only protects memory that is dynamically allocated using previously mentioned functions such as `malloc`. There is no reason that extensions could not also provide protection to all data regions including the code, initialized data, and BSS sections.

As LIBSDC 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 to choose a *max-unlocked* value that does not needlessly leave more pages than necessary in an *unlocked* state when not being utilized.

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 LIBSDC is never notified of page accesses to buffers. Data written through DMA would appear as corruption to LIBSDC because the changes were made while the data pages written were in a *locked* state.

3 Implementation

LIBSDC protects memory from SDCs by comparing last known good hashes of virtual memory pages with a hash of their current data upon page access by an application. Therefore it is critical that LIBSDC be able to receive notification when a page is being accessed by an application. To achieve this, LIBSDC uses the `mprotect` system call to modify page permissions and take away read and write access. By installing a signal handler for `SIGSEGV` (segmentation fault), LIBSDC is notified by the operating system any time a *locked* page (one without read/write permissions) is accessed. Upon notification, LIBSDC uses an internal table to verify that the page being accessed is one that it intends to protect. If it is, then verification is performed by taking a hash of the current page and comparing it to the last known good hash which is stored in LIBSDC's table. After verification, the page's read and write permissions are restored using mprotect before returning control to the user application upon exiting the signal handler.

Internally, the table that LIBSDC uses to store information on pages is compromised of several fields:

– A status flag to indicate locked, unlocked, or not managed by LIBSDC
– Storage for the page's last known good hash
– Pointers to indicate which pages were accessed for use as a first-in-first-out queue

Of particular interest of the LIBSDC's table fields are the FIFO pointers. In order to maintain a fair policy for evicting *unlocked* pages when the application needs to access a page that is not currently available, LIBSDC maintains FIFO ordering so that the oldest pages in the table are evicted first. Unfortunately once a page is in the *unlocked* state it is not possible to track accesses to the page until it is again locked. For this reason, the FIFO queue is based on the order of unlocking, and while it may not exactly mirror an application's data access patterns, it should be similar.

Each locked page's hash storage is tunable to accommodate the size of whichever hashing algorithm is used. Additional fields can also be added to accommodate storage for other needs such as ECC codes.

3.1 Handling User Pointers with System Calls

The use of a `SIGSEGV` handler allows the application's data it depends on to automatically transition from the *locked* state to the *unlocked* on demand during execution. Unfortunately, any system calls that are executed in kernel space do not enjoy this luxury as kernel space does not call the `SIGSEGV` handler during a page fault in a system call. System calls that attempt to access user space pointers will fail unpredictably if proper page permissions are not applied prior to the system call occurring. Therefore all system calls that accept user space pointers require hooking in order to unlock memory regions that the kernel is likely to access during the system call.

While in many cases it is possible to override GLIBC calls between application linking/loading and replace them with wrappers that unlock any pointers present, the GLIBC implementation may make system calls directly within itself instead of using your wrapper. For this reason it is essential that all system calls are wrapped no matter their source. For simplicity, our LIBSDC prototype makes a clone process of the original using the `clone` system call with `CLONE_VM` as a parameter to share address spaces, and then uses the `ptrace` system call to trace the application as it executes in order to receive notification of all system calls occurring. The ptrace interface is provided as part of the Linux kernel and allows a process to intercept all system calls and signals that another process generates.

It should be noted that there are other less portable solutions that may accomplish system call hooking, but would require extensive per-platform work such as binary rewriting to hook system calls or specialized kernel modules that wrap system calls. Our prototype's goal was to provide a platform for gauging the viability and costs of SDC protection through hashing and page protection while avoiding writing a complex platform specific system call hooking scheme that would not add to the research contributions.

4 Results

To gauge the overheads and demonstrate the effects of tuning the *max-unlocked* value, the HPCCG Mantevo Miniapp[8] was run with matrix size of 768x8x8

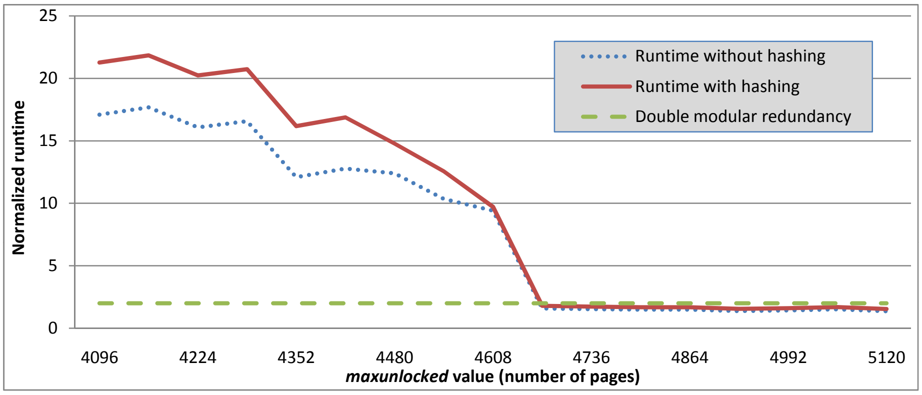

Fig. 1. Normalized execution run-times of LIBSDC with HPCCG

scaled over 256 processes. The compute nodes used consisted of 2-way SMPs with AMD Opteron 6128 (Magny-Cours), 32GB of memory per node, and a 40Gb/s Infiniband interconnect.

In Figure 1 we compared normalized execution time vs. the *max-unlocked* value to demonstrate the effects of LIBSDC on an application. The baseline execution time was taken by running HPCCG without LIBSDC performing any `mprotect` calls and by default leaving all memory in an *unlocked* state. As a comparison, the dashed line with a constant normalized time of 2 demonstrates the overhead of double modular redundancy. LIBSDC's overheads are shown with the dashed line indicating the run-times without hashing and the solid line indicating the run-times with hashing.

The choice of a range for *max-unlocked* between 4096 and 5120 is due to the maximum working-set of pages residing near the middle of that range at around 4672 unlocked pages. As depicted in Figure 1, there is a dramatic drop in the normalized run-time when we tune LIBSDC to use a *max-unlocked* value that corresponds well to the active number of working pages. From the *max-unlocked* range of 4672 to 5120, the normalized execution time falls from 1.79 to 1.53 respectively, which shows good improvement over even double modular redundancy. Although not shown, in the poorly tuned ranges below 4096 a normalized run-time of 21 or greater was observed.

For the results reported above, the average time spent calculating hashes during execution is 15%.

It is important to note that the performance of LIBSDC's hashing is highly dependent on both the hashing algorithm used and on the way it is computed. Although we chose to use SHA-1 computed on the CPU, research on computing hashes of pages using GPUs[2] has demonstrated that hashing performance on GPUs greatly outperforms CPUs. This research indicates that applications requiring page hashes should not consider the hashing itself to be a bottleneck.

We also find that the reason for the substantial overhead incurred with LIBSDC for a *max-unlocked* value less than the working-set of pages is due to our use of the `ptrace` system call. `ptrace` is known to have performance penalties due to frequent context switching on each system call and each received signal as well as generating O.S. noise. This is worsened because each page unlock is intercepted by `ptrace` during execution. While our prototype shows good performance for a well tuned *max-unlocked* value, we expect that a production version of LIBSDC would not use `ptrace` to intercept system calls. This would also result in better performance for applications running with a well tuned *max-unlocked* value, too.

5 Related Work

Similar to LIBSDC, another approach [10] that is transparent to the application achieves software-implemented error detection and correction using background scrubbing combined with software calculated ECC to periodically validate all memory and correct errors if possible. While this approach and LIBSDC are both entirely transparent to the application, LIBSDC 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 noise during execution.

Other techniques involve modifying either the application source or the compiled form of the application to generate redundancy in data, instructions, or both:

Source-to-source transformation techniques [6] have been investigated that generate a redundant copy of all variables and code at the source level. Throughout the transformed source code there 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. LIBSDC 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 LIBSDC), and can be instructed to protect any region of memory at run-time.

Duplicated instructions is another proposed technique to increase SDC resilience in software. EDDI [4] 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. LIBSDC 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 the EDDI have been proposed [7] 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 [5]. Unfortunately control-flow integrity verification does not necessarily protect against SDCs that only alter data without affecting the execution path of an application.

6 Conclusions and Future Work

In this paper we have presented a prototype implementation of the silent data corruption detection library, LIBSDC. LIBSDC is a transparent, tunable library that provides page-level protection against DRAM memory corruption. Initial results show that this library is capable of providing SDC protection to parallel HPC applications at a cost less than that of double modular redundancy.

Using LIBSDC, we were able to protect all dynamically allocated memory regions of the HPCCG application with a 53% increase in run-time over a baseline that lacked any SDC protection. Provided with hints from the application on which regions of memory to protect, LIBSDC's coverage can be tuned for an application, therefore further reducing run-time overheads.

The results of this work are very promising, but further work is needed. One considerable source of run-time overhead in our prototype implementation is the `ptrace` mechanism. Once again, we use `ptrace` to intercept system calls and ensure proper memory tracing and tracking is performed before the OS performs the call. We believe that we can remove `ptrace` from the library and provide an optimized system call wrapper to intercept these calls, though special care must be taken in these wrapper functions as issues such as reentrancy become critical to performance and correctness. In addition, we are investigating mechanisms to enable LIBSDC to use a software-based error-correcting code side-by-side with its current hash-based detection mechanisms. Use of these more advanced error-correcting codes, for example codes that can correct double-bit errors, will provide a level of protection beyond what is currently available today in enterprise-class hardware.

References

1. 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 (1984)
2. Ferreira, K.B., Riesen, R., Brighwell, R., Bridges, P., Arnold, D.: libhashckpt: Hash-Based Incremental Checkpointing Using GPU's. In: Cotronis, Y., Danalis, A., Nikolopoulos, D.S., Dongarra, J. (eds.) EuroMPI 2011. LNCS, vol. 6960, pp. 272–281. Springer, Heidelberg (2011)
3. Huang, K.H., Abraham, J.: Algorithm-based fault tolerance for matrix operations. IEEE Transactions on Computers C-33(6), 518–528 (1984)
4. Oh, N., Shirvani, P., McCluskey, E.J.: Error detection by duplicated instructions in super-scalar processors. IEEE Transactions on Reliability 51(1), 63–75 (2002)
5. Oh, N., Shirvani, P., McCluskey, E.: Control-flow checking by software signatures. IEEE Transactions on Reliability 51(1), 111–122 (2002)

6. Rebaudengo, M., Reorda, M., Violante, M., Torchiano, M.: A source-to-source compiler for generating dependable software. In: Proceedings of First IEEE International Workshop on Source Code Analysis and Manipulation 2001, pp. 33–42 (2001)
7. 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, CGO 2005, pp. 243–254. IEEE Computer Society, Washington, DC (2005), http://dx.doi.org/10.1109/CGO.2005.34
8. Sandia National Laboratory: Mantevo project home page (June 2011), https://software.sandia.gov/mantevo
9. 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, SIGMETRICS 2009, pp. 193–204. ACM, New York (2009), http://doi.acm.org/10.1145/1555349.1555372
10. Shirvani, P., Saxena, N., McCluskey, E.: Software-implemented edac protection against seus. IEEE Transactions on Reliability 49(3), 273–284 (2000)