SAND2011-4448C

Cooperative Application/OS DRAM Fault Recovery

Patrick G. Bridges[1]*, Mark Hoemmen[2], Kurt B. Ferreira[1,2], Michael A. Heroux[2], Philip Soltero[1], and Ron Brightwell[2]

[1] Department of Computer Science
University of New Mexico
Albuquerque, NM 87131
{bridges,kurt,psoltero}@cs.unm.edu
[2] Sandia National Laboratories** Albuquerque, NM 87123
{mhoemme,kbferre,maherou,rbbrigh}@sandia.gov

Abstract. Exascale systems will present considerable fault-tolerance challenges to applications and system software. These systems are expected to suffer several hard and soft errors per day. Unfortunately, many fault-tolerance methods in use, such as rollback recovery, are unsuitable for many expected errors, for example DRAM failures. As a result, applications will need to address these resilience challenges to more effectively utilize future systems. In this paper, we describe work on a cross-layer application / OS framework to handle uncorrected memory errors. We illustrate the use of this framework through its integration with a new fault-tolerant iterative solver within the Trilinos library, and present initial convergence results.

1 Introduction

Proposed exascale systems will present extreme fault tolerance challeges to applications and system software. In particular, these systems are expected to suffer soft or hard errors at least several times a day. Such errors include uncorrectable DRAM failures, I/O system failures, and CPU logic failures. Unfortunately, fault-tolerance methods currently in use by large-scale applications, such as roll-back recovery from a checkpoint, may be unsuitable to address the challenges of exascale computing. As a result, applications will need to address resilience challenges previously handled only by the hardware, OS, and run-time system if they want to utilize future systems efficiently. Unfortunately, few interfaces and mechanisms exist to provide applications with useful information on the faults and failures that affect them. This is particularly true of DRAM failures, one of the most common failures in current large-scale systems [19], but for which only low-level models of failure and recovery currently exist.

* This work was supported in part by a faculty sabbatical appointment from Sandia National Laboratories and a grant from the U.S. Department of Energy and a grant from DOE Office of Science, Advanced Scientific Computing research, under award number DE-SC0005050, program manager Sonia Sachs.

** 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.

In this paper, we describe work on a collaborative application / OS system to handle uncorrected memory errors. We begin by reviewing the basics of DRAM memory failures and how they are handled in current systems. We then discuss a specific model of memory failures that we are examining, how the application, OS, and hardware can interact to provide this failure model, and how applications recover in this scenario. Based on this, we then present a simple OS and hardware interface to provide the information to applications necessary to handle these errors, and we outline an implementation of this interface. We illustrate the use of this interface through its integration with a new fault-tolerant iterative linear solver implemented with components from the Trilinos solvers library [9], and present initial convergence results showing the viability of this recovery approach. Finally, we discuss related and future work.

2 DRAM Failures

DRAM memory modules are one of the most plentiful hardware items in modern HPC systems. Each node may have dozens of DRAM chips, and large systems may have tens or hundreds of thousands of DRAM modules. The combination of the quantity and the density of the information they store makes them particularly susceptible to faults. As a result, most HPC systems include some built-in hardware fault tolerance for DRAM. The most common hardware memory resilience scheme has the CPU memory controller write additional checksum bits on each block of data written (128-bit blocks are used on modern AMD processors, for example). The controller uses these bits to detect and correct errors reading these blocks of data back into the CPU. Most modern codes use Single-symbol[3] Error Correction and Double-symbol Error Detection (SEC-DED) schemes, allowing them to recover from the simplest memory failures and at least detect more complex (and less frequent) ones.

Recent research has shown that uncorrectable errors (e.g., double-symbol errors) are increasingly common in systems with SEC-DED memory protection [19], with uncorrectable DRAM errors occuring in up to 8% of DIMMS per year. Such errors result in a machine check exception being delivered to the operating system, which then generally logs the error and either kills the application or reboots the system depending upon the location of the error in memory. Some systems exist for recovering from such errors, for example in Linux when they occur in memory used for caching data or owned by a virtual machine [13], but these systems are much too low-level to be useful to application developers.

3 Fault and Recovery Model

3.1 Fault Model

Because of their frequency, which is only expected to increase in future systems, it is particularly important that applications be able to recover from DRAM

[3] A symbol in modern DRAM systems typically comprises 4 or 8 bits of data.

memory failures that hardware schemes cannot correct. Because these failures can potentially destroy important application data, however, recovering from them can be challenging. To address this issue, we propose a specific model for memory failures that we believe is relevant to a range of applications, discuss the application, OS, and hardware requirements of providing this failure model, and discuss how applications can recover given this failure model.

The basic model of failures we explore in this work is of *detected, transient* memory failures. In this model, applications designate portions of DRAM memory may occasionally return incorrect values, these values are expected to revert to their original value in a finite amount of time, and the application is notified when it has consumed a potentially corrupted memory location.

While most DRAM memory errors are not in fact transient, the application, OS, and hardware can cooperate to provide transience to DRAM memory failures. Specifically, hardware-level memory error detection and either application or system-level checkpointing [6] can work together to provide the appearance of transience. This is particularly effective when the application and system rarely modify the data being protected. For example, iterative methods for solving linear systems $Ax = b$ generally do not modify the matrix A and any preconditioner(s), which we exploit later in this work.

3.2 Recovery Model

To handle the failures discussed above, we present a recovery model oriented towards iterative numerical methods where the algorithm can continue to converge past faults that temporarily perturb its state. Because DRAM failures caused by reading corrupted memory can be signaled by the hardware both synchronously (due to an algorithm reading corrupted memory) and asynchronously (from the memory scrubber), we focus on a model in which the algorithm running at the time of a failure temporarily patches failures to allow progress to continue to be made and then fully recovers from the detected error later.

Specifically, our recovery model comprises the following steps:

1. The application designates to the operating and runtime system the portions of its memory in which it can tolerate a transient memory error. Errors in portions of memory not so designated are deemed fatal and cause application termination.
2. Upon receiving notification of an uncorrectable memory error in designated memory, the OS signals the application that an error has occurred at a specified address.
3. The application performs immediate but potentially partial fixes on the failed memory sufficient to allow the current calculation to continue, possibly with some degradation of accuracy.[4]

[4] For example, a solver may replace corrupted matrix entries with averages of their uncorrupted neighbors.

```
/* Register callback for handling failure in specific allocation of
 * failable memory at a specified byte offset and length. arg is an
 * opaque user-supplied argument. */
typedef void (*memfail_callback_t)( void *allocation, size_t off,
                                    size_t len, void *arg);
void memfail_recover_init( memfail_callback_t cb, void *arg );

/* Allocate resp. free failable memory */
void * malloc_failable( size_t len );
void free_failable( void *addr );
```

Fig. 1. Application / Library interface to handle DRAM memory failures

4. The application records that an error has perturbed the current calculation, to prevent it from incorrectly succeeding and returning incorrect values (e.g., if the convergence metric has been corrupted).
5. The application returns control resumes application execution at the instruction that was executing when the failure was detected.
6. At a well-defined point (e.g., every few iterations), the system or application recovers the corrupted memory using, for example, an application-specific recovery technique or a local checkpoint.
7. The application records that corrupted values have been fixed, potentially allowing it to complete after an additional successful iteration.

4 Application / OS Interface

4.1 Design

We have designed an application / OS interface to support the fault and recovery models described in Section 3, and implemented a library to provide this interface. Our key design goals were to provide a simple interface for applications and algorithmic libraries, and to support existing OS-level interfaces to handling memory errors such as those provided by Linux.

This application level of this interface, shown in Figure 1, focuses on run-time memory allocation. In particular, the interface provides the application with separate calls for allocating *failable memory* – memory in which failures will cause notifications to be sent to the application. These calls work like `malloc()` and `free()`. In addition, the application also registers a callback with the library. The callback is called once for every active allocation when the library is notified by the OS of a detected but uncorrected memory fault in that allocation.

In addition to this interface, we also provide a simple producer-consumer bounded ring buffer that the application can use to queue up a sequence of failed allocations when it is notified. This ring buffer is non-blocking and atomic to allow asynchronous callbacks from the library to enqueue failed allocation that will be fully recovered at the end of an allocation. The application determines the size of this buffer when it is allocated; the number of entries needed must be sufficient to cover all of the allocations that could plausibly fail during a single iteration. For applications with relatively few failable allocations, this should be a minimal number of entries.

At the OS level, the library first notifies the operating system that it wishes to receive notifications of DRAM failures, either in general or in specific areas of its virtual address space depending upon the interface provided by the operating system. Second, the library keeps track of the list of failable memory allocated by the application so that it can call the application callback for each failed allocation when necessary. Finally, the library handles any error notifications from the operating system (e.g., using a Linux `SIGBUS` signal handler) and performs OS-specific actions to clear a memory error from a page of memory if necessary prior to notifying the application of the error.

4.2 Implementation

We added support for handling signaled memory failures as described in the previous section to an existing incremental checkpointing library for Linux, libhashckpt [8]. We chose this library it helps track application memory usage, and provides checkpointing functionality to recover from memory failures for applications that cannot. Its ability to trap specific memory accesses eases the testing of simulated memory failures, as described later in Section 4.3.

The modified version of the library adds the application API calls listed previously in Figure 1, with the failed memory allocator using `malloc()` to allocate and free memory. This allocator also keeps a data structure sorted by allocation address of failable memory allocations.

Linux notifies the library of DRAM memory failures, particularly failures caught by the memory scrubber using a `SIGBUS` signal that indicates the address of the memory *page* which failed. The library then unmaps this failed page using `munmap()`, maps in a new physical page using `mmap()`, and calls the application-registered callback with appropriate offset and length arguments for every failable application allocation that overlapped with the page that included the failure.

Note that Linux currently only notifies the application of DRAM failures detected by the memory scrubber. When the memory controller raises an exception caused by the application attempting to consume faulty data, Linux currently kills the faulting application. In addition, Linux only notifies applications of the *page* that failed and expects the application to discard the entire failed page. This approach is overly restrictive in some cases, as the hardware notifies the kernel of the memory bus line that failed, and some memory errors are soft and could be corrected simply by rewriting the failed memory line.

4.3 Testing Support

To provide support for testing DRAM memory failures, we added support to the incremental checkpointing library for simulating memory failures. In particular, we added code that randomly injects errors at a configurable rate into the application address space and uses page protection mechanisms, i.e., `mprotect()`, to signal the application with a `SIGSEGV` when it touches a page to which a simulated failure has been injected. The library then catches `SIGSEGV` and proceeds as if it had received a memory failure on the protected page. We also implemented a simulated memory scrubber in the library which can asynchronously inject memory failures into the application by signalling the library when it scrubs a memory location at which a failure has been simulated.

5 Fault-Tolerant GMRES

We have used the interface described in Section 4 to implement Fault-Tolerant GMRES (FT-GMRES), an iterative method for solving large sparse linear systems $Ax = b$. FT-GMRES computes the correct solution x even if the system experiences uncorrected faults in both data and arithmetic [10]. The algorithm will always either converge to the right answer, or (in rare cases) stop and report immediately to the caller if it cannot make progress. The algorithm accomplishes this by dividing its computations into *reliable* and *unreliable* phases. Rather than rolling back any faults that occur in unreliable phases, as a checkpoint / restart approach would do, FT-GMRES rolls forward and progresses through any faults in unreliable phases. FT-GMRES can also exploit fault detection, including but not limited to the ECC memory fault notification discussed above, in order to decide whether to accept the result of each unreliable phase.

FT-GMRES is based on the Flexible GMRES (FGMRES) algorithm of Saad [16]. FGMRES extends the Generalized Minimal Residual (GMRES) method of Saad and Schultz [18], by “flexibly” allowing the preconditioner (also called the “inner solve”) to change in every iteration. The key observation behind FT-GMRES is that flexible iterations allow successive inner solves to differ arbitrarily, even unboundedly. This suggests modeling memory or arithmetic faults in the inner solves as “different preconditioners.” Taking this suggestion results in FT-GMRES. The algorithm uses any existing solver to “precondition” an FGMRES-based outer iteration. Our experiments use as the existing solver an iterative method such as GMRES with its own preconditioner, though it may be any algorithm (direct or iterative, dense or sparse or structured) that solves linear systems.

FT-GMRES expects that inner solves do most of the work, so inner solves run in the less expensive unreliable mode. The matrix A, right-hand side b, and any other inner solver data may change arbitrarily, and those changes need not even be transient. However, each outer iteration of FT-GMRES must run reliably, and requires a correct copy of the matrix A, right-hand side b, and additional outer solve data (the same that FGMRES would use). Since FT-GMRES expects only a small number of outer iterations, interspersed by longer-running inner solves, we need not store two copies (unreliable and reliable) of A and b in memory. Instead, we can save them to a reliable backing store, or even recompute them. With fault detection, we can avoid recovering or recomputing these data if no faults occurred, or even selectively recover or recompute just the corrupted parts of the critical data.

6 Initial Results

We have implemented FT-GMRES using components of the Trilinos solvers library [9], in particular Tpetra sparse matrices and vectors, Belos iterative linear solvers (FGMRES for the outer iterations, and GMRES for the inner iterations), and Ifpack2 incomplete factorization preconditioners. Trilinos already provides a full hybrid-parallel (MPI + threads) implementation; we chose for simplicity to limit initial experiments to a single MPI process. We only needed to modify Trilinos to use the unreliable memory allocation interface described above. Our entire solver prototype uses only 2500 lines of C++ code. For experiments, we used the random fault injection feature of the incremental checkpointing library. Detection of faults works entirely independently of fault injection, so the same code would allow us to test actual hardware faults.

We tested FT-GMRES using the development (10.7) version of Trilinos, on a Intel Xeon X5570 (8 cores, 2.93 GHz) CPU with 12 GB of main memory. We chose for our test matrix an ill-conditioned Stokes partial differential equation discretization `Szczerba/Ill_Stokes` from the University of Florida Sparse Matrix Collection (UFSMC) [4]. It has 25,187 rows and columns, 193,216 stored entries, and an estimated 1-norm condition number of 4.85×10^9. For initial experiments, we chose a uniform $[-1, 1]$ pseudorandom right-hand side.

We ran FT-GMRES with 10 outer iterations. Each inner solve used 50 iterations of standard GMRES (without restarting), right-preconditioned by ILUT (see e.g., Saad [17]) with level 2 fill, zero drop tolerance, 1.5 relative threshold and 0.1 absolute threshold. (These are not necessarily reasonable ILUT parameters, but they ensure a valid preconditioner for the problem tested.) We compared FT-GMRES with standard GMRES both with and without restarting: 500 iterations of each, restarting if applicable every 50 iterations. (This makes the memory usage of the two methods approximately comparable.) We set no convergence criteria except for iteration counts, so that we could fully observe the behavior of the methods. Our initial experiments use random fault injection at a rate of 1000 faults per megabyte per hour, which is high but demonstrates the solver's fault-tolerance capabilities. Faults were allowed to occur in floating-point data belonging to the matrix and the ILUT preconditioner. Furthermore, to demonstrate the value of algorithmic approaches, our restarted GMRES implementation imitated FT-GMRES by also refreshing the matrix and ILUT preconditioner from reliable storage before every restart cycle. (We optimized by not refreshing if no memory faults were detected.)

Figure 2 shows our convergence results. FT-GMRES' reliable outer iteration makes it able to roll forward through faults and continue convergence. The fault-detection capabilities discussed earlier in this work let FT-GMRES refresh unreliable data only when necessary, so that memory faults appear transient to the solver.

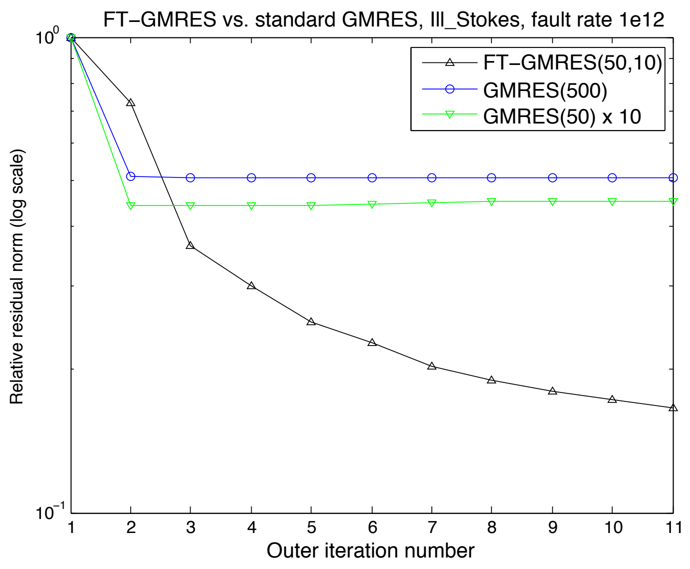

Fig. 2. FT-GMRES (10 outer iterations, 50 inner iterations each), 500 iterations of nonrestarted GMRES, and 10 restart cycles (50 iterations each) of restarted GMRES. (Down is good.)

7 Related Work

There has been a wide range of research on application and OS techniques for recovering from faults in HPC systems. This includes both algorithmic work on allowing specific numeric codes to run failures, and OS-level work on handling memory faults both transparently and directing to the application. In the remainder of this section, we describe related work in both areas.

7.1 DRAM Error Handling

DRAM errors have been intensively studied in both HPC and other large computing systems recently. Recent research has characterized both the frequency of such errors in large scale systems [19], as well as the low-level patterns with which they occur [14].

OS-level handling of such faults has generally been either very limited or used very heavyweight solutions. Linux and other operating systems, for example, provide low-level techniques for handling, logging, and notifying the application of such errors [13]. These techniques generally terminate the application, potentially invoking higher-level recovery systems based on, for example, checkpointing or redundancy. Some systems have attempted to provide additional protection against memory faults both on CPUs [5] and GPUs [15], though with substantial cost.

7.2 Fault Tolerant Algorithms

Most work in numerical linear algebra on fault tolerant algorithms falls in the category of *algorithm-based fault tolerance* (ABFT) (see e.g., [12]). ABFT encodes redundant data into the matrices and vectors themselves, such that data owned by failed parallel processors can be recomputed. Recent results using this technique show this method can be used with a very low performance overhead [3, 1]. Other authors have empirically investigated the behavior of iterative solvers when soft faults occur (e.g., [1, 11]).

Inner-outer iterations with FGMRES have been used as a kind of iterative refinement in mixed-precision computation [2], but as far as we know, this work is the first to use flexible iterative methods for reliability and robustness against possibly unbounded errors. *Inexact Krylov methods* generalize flexible methods by allowing both the preconditioner and the matrix to change in every iteration [20, 7]. However, inexact Krylov methods require error bounds, and thus cannot be used to provide tolerance against arbitrary data and computational faults when applying the matrix A.

8 Conclusions and Future Work

In this paper we presented a cooperative cross-layer application / OS framework for recovering from DRAM memory errors. This framework allows the application to allocate *failable* memory and provides notification and callback mechanisms for failures that occur within these failable allocations. We described the fault and recovery model for this framework. Finally, we presented initial results of this framework using the new fault-tolerant linear solver FT-GMRES, and showed that the solver is able to converge in the presence of memory failures.

These initial results are promising, but more work is needed. Larger-scale studies are need to better characterize the cost of this framework at scale. Additionally, we are in the process of identifying more algorithms that can benefit from this DRAM failure framework.

References

1. Bronevetsky, G., de Supinski, B.: Soft error vulnerability of iterative linear algebra methods. In: Proceedings of the 22nd Annual International Conference on Supercomputing. pp. 155–164. ICS '08, ACM, New York, NY, USA (2008), `http://doi.acm.org/10.1145/1375527.1375552`
2. Buttari, A., Dongarra, J., Kurzak, J., Luszczek, P., Tomov, S.: Computations to enhance the performance while achieving the 64-bit accuracy. Tech. Rep. UT-CS-06-584, University of Tennessee Knoxville (November 2006), lAPACK Working Note #180
3. Chen, Z., Dongarra, J.: Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In: Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International (April 2006)

4. Davis, T.A., Hu, Y.: The University of Florida Sparse Matrix Collection. ACM Trans. Math. Softw. (to appear) (2011), \url{http://www.cise.ufl.edu/research/sparse/matrices}
5. Dopson, D.: SoftECC: A System for Software Memory Integrity Checking. Master's thesis, Massachusetts Institute of Technology (September 2005)
6. Elnozahy, E.N., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Computing Surveys 34(3), 375–408 (2002)
7. van den Eshof, J., Sleijpen, G.L.G.: Inexact Krylov subspace methods for linear systems. SIAM J. Matrix Anal. Appl. 26(1), 125–153 (2004)
8. Ferreira, K.B., Riesen, R., Brightwell, R., Bridges, P.G., Arnold, D.: Libhashckpt: Hash-based incremental checkpointing using GPUs. In: Proceedings of the 18th EuroMPI Conference. Santorini, Greece (September 2011)
9. Heroux, M.A., Bartlett, R.A., Howle, V.E., Hoekstra, R.J., Hu, J.J., Kolda, T.G., Lehoucq, R.B., Long, K.R., Pawlowski, R.P., Phipps, E.T., Salinger, A.G., Thornquist, H.K., Tuminaro, R.S., Willenbring, J.M., Williams, A., Stanley, K.S.: An overview of the Trilinos project. ACM Trans. Math. Softw. 31(3), 397–423 (2005)
10. Heroux, M.A., Hoemmen, M.: Fault-tolerant iterative methods via selective reliability. Tech. Rep. SAND2011-3915 C, Sandia National Laboratories (2011), available at http://www.sandia.gov/~maherou/
11. Howle, V.E.: Soft errors in linear solvers as integrated components of a simulation. In: Presented at the Copper Mountain Conference on Iterative Methods, Copper Mountain, CO, April 9 (2010)
12. Huang, K.H., Abraham, J.A.: Algorithm-based fault tolerance for matrix operations. IEEE Transactions on Computers C-33(6) (June 1984)
13. Kleen, A.: mcelog: memory error handling in user space. In: Proceedings of Linux Kongress 2010. Nuremburg, Germany (September 2010)
14. Li, X., Huang, M.C., Shen, K., Chu, L.: A realistic evaluation of memory hardware errors and software system susceptibility. In: Proceedings of the 2010 USENIX Annual Technical Conference (USENIX'10). Boston, MA (June 2010)
15. Maruyama, N., Nukada, A., Matsuoka, S.: A high-performance fault-tolerant software framework for memory on commodity GPUs. In: Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on. pp. 1 –12 (april 2010)
16. Saad, Y.: A flexible inner-outer preconditioned GMRES algorithm. SIAM J. Sci. Comput. 14, 461–469 (1993)
17. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM, Philadelphia, second edn. (2003)
18. Saad, Y., Schultz, M.H.: GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM J. Sci. Statist. Comput. 7, 856–869 (1986)
19. Schroeder, B., Pinheiro, E., Weber, W.D.: DRAM errors in the wild: a large-scale field study. Communications of the ACM 54, 100–107 (February 2011), http://doi.acm.org/10.1145/1897816.1897844
20. Simonici, V., Szyld, D.B.: Theory of inexact Krylov subspace methods and applications to scientific computing. SIAM J. Sci. Comput. 25(2), 454–477 (2003)