# Memory reliability and performance degradation

Hunting rabbits with an elephant gun?

Benjamn A. Allan
Scalable and Secure Computing Research Department
Sandia National Laboratories
Livermore, CA USA
baallan@sandia.gov

***Abstract*— We [1] review several approaches to identifying silent memory error events needed to estimate rates at scale, and conclude that the easy approaches are all apparently blocked by pragmatic concerns. We aim to provoke a discussion on what other approaches we may have overlooked and on what could be done to shift the pragmatic barriers. This particular question fits into the larger category of continual monitoring at scale for hardware performance changes, but is unique in that the benchmarking process necessarily consumes much wall time but little in the way of processor cycles, bandwidth, storage space, or energy.**

***Keywords—Linux; fault detection; high performance computing; memory management***

## I. INTRODUCTION

On the journey to extreme scale the threat of transient bit flips (soft errors) corrupting simulation data spread across millions of electronic components forces us to better quantify the risk of this corruption. Network and storage engineers have long dealt with data corruption, but for most of the software community the issue is new. Numerous papers on fault tolerant algorithms to handle explicit hardware failure have been published and also on the need for fault oblivious algorithms [1]. Tools that estimate the impact of single errors on specific code have been produced [2]. Assuming no game-changing memory technology will appear at commodity scale, processor designers have proposed using radiation-hardened cores including a small amount of highly reliable memory to protect operating systems from errors, explicitly leaving application developers to protect their bulk data.

Unfortunately, short of triple-redundant computation (and memory) schemes, no general solution has been proposed to the silent corruption of address (pointer) data. While a corrupted pointer will often cause a crash in a small-memory program (by violating address protection rules or pointing to evidently nonsensical data), the odds are very good that a corrupted pointer in a large-memory program will still point to program-owned memory and to data of the same type and even similar values.

## II. SOFTWARE APPROACHES

If we cannot afford high levels of hardware redundancy, we must have a software tool set for measuring the real rates of silent errors at scale so we can appropriately calibrate the level of effort spent on developing software-level data redundancy. Laboratory (neutron bombardment) models of failure rates may not adequately characterize large, aging machines as fielded. To quantify the risk of independent events creating an undetectable multi-bit flip, we need to quantify the frequency of single events on the deployed platform. Counts of corrections and notifications of uncorrectable multi-bit errors may be difficult or impossible to obtain, depending on the architecture and software stack.

Silent errors (three-bits or more on today's ECC SRAM or DRAM) can only be detected at the software level (if we assume the double error correction schemes will remain confined to RAM on the processor dies). When an inconsistency is found, attribution of the cause is still difficult. Was the error in bulk RAM, address or cache logic, or processor arithmetic (assuming correct software)? Separate quantification of error rates in each subsystem must be considered, but from the application view point, the overall rate may be sufficient for budgeting software development work or for selecting among more and less reliable nodes within a cluster of nodes having known component silent error rates.

Let us consider the ways software could be deployed to quantify the location and rates of these rare events and characterize the nodes (or memory-heavy cores) of an aging future machine. These methods have similarities to work by others on continually characterizing the performance of deployed co-processors, with the key difference that most performance benchmarks can measure the needed rates quickly (in just a few seconds to a few minutes).

[1] Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000. Report SAND2014-18202 C

### A. Dedicated (burn-in) Benchmarking

Correctible soft errors may appear as performance degradation (extra ECC operations) only. Benchmarking done before a machine is put into production may identify defective parts, but such benchmarks are rarely repeated during production to detect aging parts with increased soft error rates. Even when performed, the results are generally reported as pass/fail; this obscures the detailed data needed to form statistical models of the soft errors seen in each hardware piece.

### B. User-level scanning application

We can easily write an HPC-cluster application which populates all available RAM with data and then periodically scans it for apparent undetected bit flips. This software needs to be aware of any hardware-based memory scrubbing and capture the address of any ECC events. Ideally, the bulk data created should be such that any corruption of the address data or handling will easily be detected as numerous words being corrupted. This approach may be applied to both bulk memory and caches, provided suitable cache management can be arranged from user-space software.

The obvious pragmatic barrier to this approach is explaining to machine owners and users why "real work" is idling in the queue system while the scanning application is seen.

### C. Idle memory scanning daemon

We can easily convert the user application just outlined into a permanent daemon job which spawns processes that consume and periodically check memory chunks. For DRAMs, the Linux kernel out-of-memory (OOM) killer [3] is easily configured to kill off these spawned processes before killing "real work". On HPC systems without swap, this allows us to regularly scan all memory not in use for productive work. The CPU (and energy) usage of these checks is easily bounded and controlled so as to be a known, small tax on real work.

There are three pragmatic barriers to this approach. First, job management systems are sometimes configured to kill an entire parallel job if any node reports an OOM condition, on the assumption that the "real work" is the only possible source of such events; changing these configurations to be more selective is socially nontrivial. Second, the stock Linux memory manager may allocate pages (if any are available) across remote memory connections in preference to invoking the OOM killer on a checker process that has consumed the physically nearest memory. This leads to large performance degradation in a large fraction of HPC applications that are memory bandwidth sensitive. If we are to use the OOM killer, we must patch the kernel to make it smarter. Third, the fine-grained control of cache behavior needed may require kernel modification to prevent scheduling other tasks on the core(s) nearest the cache being tested.

### D. Idle memory scanning kernel thread

If patching the kernel is required, we can do better than modifying the OOM killer target selection process. With significant effort, we can have a kernel thread scan idle memory for silent errors. This thread has the side effect of emulating hardware-based memory scrubbing. As with the scanning daemon, costs can be bounded. This approach would provide the highest quality and largest possible amount of information without interfering with "real work". There are two pragmatic barriers here. First, such a patch is highly likely to be rejected by the main line Linux kernel developers as being of interest to only a very small HPC community. Second, using such a patch will require, in most cases, acceptance and deployment by the platform vendor into a supported kernel.

## III. TESTING THE SOFTWARE APPROACHES

We have implemented approaches B and C. While the codes work and the OOM-killer run flawlessly on desktop machines, we run into the pragmatic barriers discussed when attempting deployment in NUMA-based production environments.

If a silent error is a rare talking rabbit to be hunted, even more rare than the black swan, then augmenting the Linux virtual memory subsystem with an advanced memory scrubbing scheme to locate and count them amounts to using an elephant gun [4] (by the time the social issues of such a kernel modification are negotiated). Before risking such a pinch, let us consider the behaviors a silent error scrubbing modification of the Linux virtual memory manager and task scheduler might have.

1. It should appear to the user as an obvious part of the kernel services, perhaps listed as [*kscrubd]* in the process table.
2. It should be enabled or disabled with a switch under /proc.
3. It should fill unused memory pages with data that can be checked with minimal computation to reveal bit errors or addressing faults. Perhaps each 64 bit word should hold its own physical address, though this may have security implications unless pages are zeroed before being handed to user processes.
4. It may scan for errors periodically, but preferably not when the processors or memory bus are near saturation. The period should be adjustable via a /proc parameter.
5. It may scan for errors as part of the page allocation process and fill as part of the deallocation process. This adds one-time per page allocation process overheads, which in bulk synchronous applications is unlikely to be noticeable compared to page use lifetime.
6. The choice or hybridization of the work strategies listed in points 4 and 5 must be controllable via /proc.
7. In NUMA environments, the processor nearest a memory should be responsible for scanning that memory, or performance impacts may be substantial.

8. If the system has provision for down-clocking memory or shutting it off entirely when usage is low, then the scrubbing scheme must be made aware of these to the degree needed to avoid false positives caused by changes in operating condition.
9. For cache scanning, we must be able to assign a core exclusively to the scanning process.

## IV. Limitations and Implications

It is quite likely that in virtual machine environments, the patterns of memory usage are sufficiently different from those of HPC systems that the most accurate approach, (D), would not be acceptable or possibly even not feasible. However, for the same models of hardware components applied in similar manners and locations we can expect the silent fault rates seen in general purpose (cloud) servers would be the same.

Neutron beam testing of ECC SRAM suggests that undetected faults may be at rates as high as 10% of the rate of detected and corrected faults [5]. Current architectures are such that 5-10% of the CPU may not be practical to protect, ensuring a source of silent errors even as ECC schemes become more sophisticated. At cloud and data center scales (hundreds of thousands of servers), this implies that there may be thousands of such errors daily across the world. While a resulting bit error in streaming video may be insignificant, silent errors in processing large memory-resident data could easily become very significant. Similarly, a silent error in the security data cache could easily change an access policy; while perhaps harder to locate than a new piece of a digital currency, a motivated and capable attacker might still succeed in breaching a server in a large and otherwise secure system by knocking often enough and getting lucky.

## V. Conclusion

To better understand and work around silent errors during the life of a production system, we must first quantify the rate of their appearance reliably and continually. We propose that such continual quantification may be accomplished opportunistically (checking idle RAM during idle cycles) without substantially affecting the power consumption or performance of a production machine. We report on early attempts to formulate a low-cost detection method at scale and solicit input on one proposed but potentially difficult to implement alternative.

## References

[1] F. Capello, A. Geist, B. Gropp, L. Kale, B. Kramer and M. Snir, "Toward Exascale Resilience," *International Journal of High-Performance Computing Applications,* vol. 23, no. 4, pp. 374-388, 2009.

[2] D. Li, J. S. Vetter and W. Yu, "Classifying Soft Error Vulnerabilities in Extreme-Scale Scientific Applications Using a Binary Instrumentation Tool," in *Supercomputing Conference*, Salt Lake City, 2012.

[3] R. Chase, "How to configure the Linux out-of-memory killer," Oracle, February 2013. [Online]. Available: http://www.oracle.com/technetwork/articles/servers-storage-dev/oom-killer-1911807.html. [Accessed 1 June 2014].

[4] C. Jones, Director, *Rabbit Fire.* [Film]. USA: Warner Brothers, 1951.

[5] J. Loncaric, Interviewee, *Personal communication.* [Interview]. 3 June 2014.