Redundant Memory Array Architecture for Efficient Selective Protection

Ruohuang Zheng Michael C. Huang
University of Rochester
{ruohuang.zheng,michael.huang}@rochester.edu

ABSTRACT

Memory hardware errors may result from transient particle-induced faults as well as device defects due to aging. These errors are an important threat to computer system reliability as VLSI technologies continue to scale. Managing memory hardware errors is a critical component in developing an overall system dependability strategy. Memory error detection and correction are supported in a range of available hardware mechanisms. However, memory protections (particularly the more advanced ones) come at substantial costs in performance and energy usage. Moreover, the protection mechanisms are often a fixed, system-wide choice and can not easily adapt to different protection demand of different applications or memory regions.

In this paper, we present a new RAIM (redundant array of independent memory) design that compared to the state-of-the-art implementation can easily provide high protection capability and the ability to selectively protect a subset of the memory. A straightforward implementation of the design can incur a substantial memory traffic overhead. We propose a few practical optimizations to mitigate this overhead. With these optimizations the proposed RAIM design offers significant advantages over existing RAIM design at lower or comparable costs.

CCS CONCEPTS

• **Computer systems organization** → **Reliability**; **Processors and memory architectures**; *Multicore architectures*; *Redundancy*;

KEYWORDS

RAIM, Memory Protection

ACM Reference format:
Ruohuang Zheng Michael C. Huang University of Rochester. 2017. Redundant Memory Array Architecture for Efficient Selective Protection. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 14 pages.
https://doi.org/10.1145/3079856.3080213

This work was supported in part by NSF under grants 1255729 and 1314734 and by SRC under Contract No. 2013-HJ-2405. The authors would like to thank Yanos Sazeides and the anonymous reviewers for their numerous valuable suggestions to improve the paper.

ISCA '17, June 24-28, 2017, Toronto, ON, Canada

ACM ISBN 978-1-4503-4892-8/17/06…$15.00
https://doi.org/10.1145/3079856.3080213

1 INTRODUCTION

A memory hardware error occurs when a memory cell loses its state or supplies a wrong value to the processor. Memory errors are sometimes caused by temporary environmental factors such as particle strikes from radioactive decay and cosmic ray-induced neutrons [29, 35, 52, 53, 55]. These errors, called *transient* errors, do not persist and are correctable by software overwrites or hardware scrubbing. Other errors are caused (at least partially) by inherent manufacturing defect, insufficient burn-in, or device aging [5]. These errors are often *non-transient* in that they tend to cause repeating symptoms as the deterioration is often irreversible. Memory errors and bit corruptions are an important threat to computer system reliability, as demonstrated by high-profile incident reports (e.g., for Sun Microsystems servers [42] and Amazon cloud computing facility [2]). Past case studies [33, 44] further suggested that these errors are significant contributing factors to whole-system failures. Managing memory hardware errors is an important component in developing an overall system dependability strategy.

Memory error detection and correction are supported in a range of available hardware mechanisms such as SECDED (single-error correction, double-error detection) ECC, memory scrubbing, Chipkill [9], active memory mirroring [12, 16], and redundant array of independent memory (or RAIM, in IBM zEnterprise system) [30]. Memory error protection carries potentially substantial costs in memory space, performance, and energy usage. For instance, the Chipkill mechanism was shown to reduce the effective memory bandwidth by 40% [10]. Furthermore, Chipkill and certain flavors of RAIM [30] spread data over multiple channels which dilutes the locality at the DRAM core and multiplies the number of banks and associated interface elements to activate (hence energy costs).

A memory error that escaped hardware memory protection is exposed to the software level. However, its corrupted memory value may or may not be consumed by software programs. Even if it is consumed, the software system and applications may continue to behave correctly if such correctness does not depend on the consumed value. In principle, different software memory areas may have unequal criticality or susceptibility to memory hardware errors. For instance, the reliability of the system software (operating system or virtual machine hypervisor) is more critical than that of application software (which may be restarted without affecting other applications in the system). Certain data such as media may be inherently more tolerant to errors and allow natural graceful degradation. In such a case, terminating the application upon detecting an uncorrectable error is not only unnecessary but counterproductive. In general, a fixed, system-wide policy to protect memory regardless of application need is inefficient in resource expenditure and suboptimal in effects. This is especially so as we move towards the cloud environment where

a diverse collection of workloads are consolidated on to the same servers.

In this paper, we propose hardware mechanisms to support selective, non-uniform memory protection. The proposed design builds on an existing proposal of RAIM (redundant array of independent memory) architecture (which will be referred to as RAIM-3). While the existing RAIM-3 architecture provides superb error protection capabilities, the cost of protection is high in both storage and energy consumption. Furthermore, the architecture requires a large number of memory channels. We propose a different organization of the RAIM design (which will be referred to as RAIM-5) to more easily facilitate selective protection, paying storage overhead only when needed and can be easily implemented in a lower-end system with a moderate number of memory channels. Additionally, we investigate a number of optimizations to reduce the high memory traffic overhead as a result of our RAIM organization. These optimizations are practical and effective, lowering the memory traffic overhead to a comparatively insignificant level.

In the rest of the paper, we will discuss background and related work (Sec. 2), the architectural design (Sec. 3), the experimental analysis (Sec. 4), a case study of applying selective protection (Sec. 5), and conclusions (Sec. 6).

2 BACKGROUND AND RELATED WORK

Memory errors are relatively rare events that require large amount of observation and/or special mechanisms to study. Many past studies have brought insights to the problems [8, 24, 25, 35, 36, 40, 41, 51, 54]. Some general observations include that memory errors are a non-trivial phenomenon and are likely to be caused by transient as well as more permanent fault mechanisms. The understanding has also led to numerous proposals of protection mechanisms.

Protection mechanisms and their costs: One category of memory errors are due to storage. In these cases, a storage cell may become defective or temporarily lose its state due to noise (energy). Another category of errors are due to the datapath from the interconnect fabric to the interface circuitry on the memory chip. All these elements can have transient or permanent errors that prevent correct data at the storage cells from reaching the processor core. Computer systems, especially high-end servers, use a whole host of protection mechanisms to combat these errors: error-correcting code (ECC) on the memory arrays, cyclic redundancy code (CRC) for the interconnect fabric, spare circuits inside memory chips and spare chips, and dynamic controls such as retry and scrubbing. SECDED uses parity-check code to support single-error correction and double-error detection. Chipkill ECC is designed to tolerate word-wise multi-bit errors such as those that occur when an entire memory device fails [9]. It can be implemented using wider ECC codes to detect and correct errors involving more bits, or by marshaling bits such that only one bit from each memory device is used to form a word. The active memory mirroring [16] stores identical data on a pair of DIMMs. If an uncorrectable error occurs in one, the data will be retrieved from the mirrored DIMM. The IBM zEnterprise system employs redundant array of independent memory (RAIM) architectures [30]. This particular implementation of RAIM is similar to RAID 3, and uses five memory channels to store data logically striped across four memory channels. Each channel also uses extra storage for ECC (8 chips for data, 1 for checksum). This system can correct a variety of errors such as that resulting from a failure of an entire memory channel plus two extra memory chips [30].

Error protection measures come with non-trivial costs in circuit, performance, and energy, some obvious, others subtle. For instance, memory mirroring provides very strong protection against various forms of errors—any error localized to one memory channel—but comes at 100% overhead in memory capacity. The RAIM implementation in [30] stores 13 parity symbols[1] for the original data of 32 symbols, carrying a 40% storage overhead.

We further recognize that stronger ECC mechanisms (with higher reliability) typically incur higher costs. For instance, modern DRAM DIMMs invariably use chips that each provide multiple (say, 4) bits per access (referred to as x4 chips). Thus, these bits share single-point-of-failures in the the access circuitry of the chip and may become erroneous simultaneously. A less powerful correction code such as SECDED will fail to correct it. Indeed, the more bits to correct, the more costly the ECC circuitry is. A Chipkill ECC [9], which can correct an error involving an entire chip failure, is often implemented by spreading the data across multiple chips and multiple memory channels. This design, however, has significant costs of its own. Having to send the same command to multiple channels to fetch the same cache line, there is significantly more downstream traffic (to DRAM) and less efficient use of the upstream bandwidth due to shorter burst lengths. The result can be a 40% reduction of the effective throughput [10]. Furthermore, spreading data over multiple channels further dilutes the locality at the DRAM core and multiplies the number of the banks and associated interface elements to activate (and hence energy costs).

A number of novel protection mechanisms have been proposed to improve the cost benefit ratio. Udipi et al. propose to spread parity code around different chips to handle single chip failure in systems with larger symbols [47]. A later proposal also separate codes needed for detection and for correction, also targeting systems with DRAM chips providing more bits [46]. COP uses memory compression to free up space for ECC overheads [37]. Bamboo uses a novel coding architecture to provide strong protection with low storage overhead. Multidimensional parity code is another example of efficient coding [13]. Parity Helix maps ECC words in a helix fashion in 3D die-stacked DRAMs so that an entire dimensional failure can be protected against with a low cost [14]. In addition to better error-correction codes, other proposals include techniques to remap defective cells [19, 34] and a novel hashing-based error detection mechanism [6].

Selective error protection: While most existing mechanisms uniformly protect memory areas in the system, limited support for selective error protection also exists. For example, the active memory mirroring [16] can be applied to a selective memory region. However, due to a lack of software error susceptibility information, its practical use is confined to protecting the IBM POWER system hypervisor in a coarse-grained fashion. Mehrara and Austin worked on selective placement of critical data for partial memory protection [31]. They analyze data read/write phases to identify the "live" time when a transient memory error would be read by the software.

[1] 1 parity symbol per data channel, 9 for the parity channel.

Khudia and Mahlke have proposed a completely automated analysis to partition computation into different categories with different protection requirements [18]. These past works complement our work by providing policies for selective protection whereas our work provides the mechanisms.

The most closely related proposal to ours is virtualized ECC [49] where the OS explicitly maps space for redundancy into physical storage. Our work offers a simpler organization of redundancy information based on RAIM and provides even stronger protection when needed.

Selective protection of CPU processing errors has also been addressed in the past [7, 26, 39, 43, 45]. CPU error protection, however, focuses on the identification of critical instructions in the code, rather than data in memory, that are most susceptible to catastrophic error consequences. These techniques cannot assess the impact of memory errors through execution (and, in particular, propagation), and therefore they cannot be directly utilized to support non-uniform memory protections. Finally, there are proposals to leverage locality to be selective in cache memory protection as a means to reduce the (storage) cost [20, 21, 50]. In these proposals, the ECC code word or the shadow copies of the more recently used cache lines are kept—displacing those from the less recently used lines. As a result, a small storage investment can protect those lines that cover a disproportionately high access frequency.

A recent study of data-intensive applications found that different regions have different tolerance to errors and that non-uniform memory protection can be effective [28]. Automated methodologies for evaluation and for software-based error recovery are also being studied [15, 27]. Finally, selective protection can help DRAM timing optimization proposals [17, 23, 38].

3 ARCHITECTURE SUPPORT AND OPTIMIZATIONS

We start with a state-of-the-art error-correcting memory architecture and explain its disadvantages in supporting selective protection (Sec. 3.1). We then describe an alternative architecture that more easily supports selective protection and discuss its costs (Sec. 3.2). We then discuss microarchitectural optimizations that mitigate these costs (Sec. 3.3).

3.1 Baseline ECC Architecture

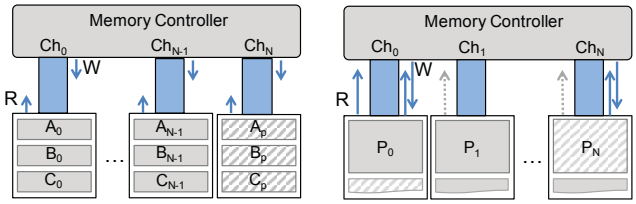
We start with a baseline (non-adaptive) design similar to [30]. The system applies RAID architecture with memory channels and is naturally referred to as RAIM. In particular, the architecture adopted is that of RAID-3 design, where a unit of protection (in this case a cache line) is split into N chunks, each stored within an independent memory channel. In addition, a parity chunk is simply the result of bitwise XOR of the N data chunks and is stored in a separate memory channel (Fig. 1). We will call this design RAIM-3.

When a cache line read occurs, all memory channels are simultaneously accessed to retrieve the individual chunks. This system can correct a variety of errors such as that resulting from a failure of an entire memory channel (including the storage devices and the communication links) plus two extra memory chips from other channels [30].

Such strong protection clearly comes with significant cost. For example, the RAIM implementation in [30] uses storage for 45 symbols for the original data of 32 symbols, carrying a 40% storage overhead. Moreover, even this level of storage overhead is attained only with a relatively large number of memory channels (five), which in itself carries non-trivial costs: Modern chips are increasingly pin-limited. Connecting a larger number of memory channels to a processor often requires an intermediate buffer chip that communicates with the processor chip with high-bandwidth-per-pin channels [48]. Such design is more costly and also increases latency (by 11 nanosecs in the case of the IBM AMB chip [48]) and energy for every access.

In addition to the significant storage overhead and the requirement of a large number of memory channels, the hardware may also spend significant time recovering from an error. For instance, one type of recovery process (called tier-2 recoveries) can take approximately 150 μsecs, or about 1 million cycles [30]. During this period, the loss of access to a memory channel degrades error correction capability and an otherwise correctable error may cause the access to be delayed after the recovery.

The issue with such a design is not cost per se. After all, robustness of mission critical application is far more valuable compared to the costs. The issue, however, is that such protection is fixed for the system regardless of the protection demand of the application, leading to not only waste but even lower quality of service. Indeed, depending on the data, the application may be able to proceed with an alternative approach of tolerating an error providing an equally acceptable response at a much lower latency and therefore provide a *higher* level of overall system robustness.



(a) RAIM-3 (b) RAIM-5

Figure 1: Operational differences between the two RAIM implementations. In RAIM-3, a cache line A is striped over multiple channels (A_0 to A_{N-1}). Reads and writes involve all channels, albeit with shorter bursts (indicated by short arrows). In our RAIM-5, a page (e.g., P_0) stays in one channel. The initial read involves only one channel (with a longer burst), but may encounter an uncorrectable error that requires additional access of remaining channels (dashed arrows). A writeback of a cache line involves reading and writing the line and its parity line. Parity pages are spread around all channels for load balancing.

3.2 Basic Architecture for Selective Protection

To give applications the choice of memory protection levels, it is more convenient to organize parity protection groups at the level of pages. A RAIM-5 architecture is a natural choice. In RAIM-5, pages at the same offset in different channels form a group (e.g., P_0

Table 1: Trade-offs between RAIM-3 and RAIM-5 architectures.

	RAIM-3	RAIM-5
Selective protection	Not very meaningful (overhead already paid for full protection)	Full flexibility in selective protection
Protection strength	Very high: can tolerate multiple chip failures and channel failures	Similar to RAIM-3
Num. of mem. ch.	Very limited choice (essentially just 4+1 channels)	Flexible (anything greater than one)
Cache line size	Smaller line sizes can lead to energy costs and wastes DRAM bandwidth	Not an issue
Overhead	Fixed overhead in storage, traffic, and energy. High energy overhead due to spread of access to all channels	Proportional to protection fraction, but high overhead for writes *in an unoptimized design*

.. P_N in Fig. 1). The $N+1$ pages in the same group can form a parity group and be exposed to software as N RAIM-protected pages or be exposed as $N+1$ pages without RAIM protection. The operating system will provide an interface for applications to allocate pages either in protected or unprotected mode. The protection mode is kept in page tables as well as in TLB entries to direct cache controller.

With the same underlying memory system, our page-level RAIM-5 organization differs from the baseline RAIM-3 design in a few ways. First, the $1N$ space overhead at the channel level is fixed in the RAIM-3 design whereas in our system the overhead applies only to protected pages. Second, RAIM-3 spreads a single line over N channels, essentially constraining N to power of 2. In practice, the choice is perhaps even more limited: having three channels (2+1) implies a 50% space overhead, which is probably too high. In contrast, RAIM-5 can choose any number greater than 1. Considering that RAIM will only be selectively applied, even a small channel number is acceptable. Third, when reading a line from the memory, a RAIM-3 system typically accesses all channels and carries a fixed $1N$ bandwidth overhead. In our system, a read to an unprotected page is the same as a system without RAIM protection. Even for a read to a protected page, the operation can still be overhead-free in the common, error-free cases. Only when the built-in error detection from a single channel (e.g., SECDED for the memory storage) detects an uncorrectable error will the RAIM protection be invoked to correct errors.[2] In summary, the design provides more cost-proportionality (in reads) by "taxing" only protected pages.

The aforementioned benefits come at a cost: writebacks are more expensive. When a cache line belonging to RAIM-protected pages needs to be written back, we need to update the corresponding line in the parity page. This requires calculation of the new parity in one of two ways: recalculating the parity from scratch or calculating incremental change due to the write. We refer to these two styles simply as RAIM-5a and RAIM-5b.

- **RAIM-5a**: Upon a writeback of a cache line, all lines belonging to the same parity group will be fetched so as to calculate the new parity. Some of these lines may already be on-chip in the cache and thus do not need to be fetched from the memory. Note that for simplicity of model, we only consider cache lines that are the same with their memory counterpart, i.e., they are not dirty. Under this model, the best case scenario is when all lines are on-chip and thus the cost for a writeback is just two writes (one for the original writeback and the other for writing the new parity). The worst case involves reading $N-1$ (N being the number of data lines in a parity group) lines and writing two.
- **RAIM-5b**: Upon a cache line writeback, we fetch the old copy of the line from memory; calculate the delta between the two versions (assuming the RAIM parity is simple XOR); apply the delta to the parity line by a read-modify-write operation. This approach always requires two memory reads and two memory writes.

The drawback is clear: writebacks incur several times the original cost. Even though they are not on the critical path and writebacks are much less frequent than reads, the extra energy and bandwidth overhead is still significant. As we will show later in Sec. 4, both models can increase overall memory traffic by several times. A simple hybrid that chooses the less expensive mode between RAIM-5a and RAIM-5b for every line only marginally reduces the overhead. We need effective optimizations to address the issue. Fortunately, there are a few practical optimizations that are very effective. But before delving into the details of the optimizations, we recap the different trade-offs between the two different RAIM architectures in Table 1.

3.3 Optimizations

To reduce writeback-induced overheads, we can reduce the frequency of writebacks and/or the cost for each occurrence. We discuss three approaches that turn out to be very effective. We will briefly mention mechanisms or variations that turn out to be ineffective (at least in our experimental framework). Quantitative analyses will be presented later in Sec. 4.

Gang writeback: The minimum overhead for a cache line writeback in our RAIM-5 is to write back the new parity line. This already amounts to 100% overhead in memory traffic. One way to save this overhead is to write back more than one cache line (belonging to the same parity group) together. This way, the update to parity is amortized over multiple lines.

Perhaps the most straightforward way to do such "gang writeback" is simply to write back all dirty lines in the parity group when one dirty line is being evicted. In such a design, the amount of memory accesses remains the same if we use RAIM-5a, as can be seen in Fig. 2. In short, if we do not write back the dirty lines, we would have to read their old version in order to calculate the new parity, which results in the same amount of traffic.

Note that this does not mean gang writeback is always for free – it is only so in RAIM-5a. The mode with the lowest overhead depends on the number of different types of lines and may be different from

[2]However, this approach relies on error detection built into a single memory channel (e.g., SECDED). If the built-in detection is unable to detect the error, then this low-overhead mode of RAIM-5 will not use RAIM to correct the error, and thus provide a bit less protection than RAIM-3. In order to achieve the same level of protection, RAIM-5 can be operated in a mode where it accesses all channels upon a memory request just like RAIM-3.

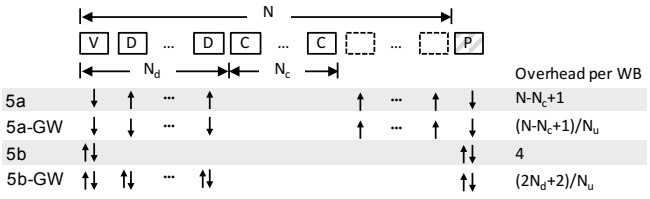


Figure 2: Illustration of cache line reads and writes needed in RAIM-5 with and without gang writebacks (GW). The lines of the parity group are organized from left to right into the dirty victim (V) causing the writeback, other dirty lines present (D), clean lines present (C), lines not present (dashed boxes), and the parity line (P). The up/down arrows indicate a read/write of the corresponding line. The overhead per useful writeback is also shown. N_u is the number of useful writebacks.

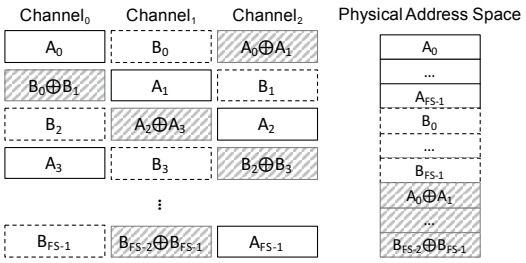


Figure 3: Mapping of physical lines to channels with corresponding parity group formations. FS stands for frame size (measured in number of lines). Each box represents a physical line. On the right, we show the physical address space; while on the left we show how lines are actually distributed in DRAM channels. A_0 through A_{FS-1} and B_0 through B_{FS-1} form two frames, and the shaded boxes form the third frame for parity data. Within a frame, A_0 and A_1 form a parity group; A_2 and A_3 form a parity group; and so on.

parity group to parity group. Furthermore, if we can accurately predict which write will be useful, we can selectively write back only those lines in RAIM-5b. In other words, there is no simple formula to get the lowest overhead. Nevertheless, we can make some general observations: empirically, the number of useful writebacks is high enough that we can make a simplistic policy of always doing gang writeback. Furthermore, if we manage to make cache lines with similar access behaviors stay in the same parity group, then the right times to write each line back will be highly correlated and thus more of the writebacks will be useful. This makes the next idea synergistic with gang writeback.

Memory interleaving: Recall that when all cache lines belonging to the same parity group are present, we can calculate the new parity when we have to write back one line. This gives the least traffic overhead. Ideally, we will like to find the group to be complete when a writeback is needed. If we map memory address in a page-interleaved fashion among the memory channels, then a particular line will be in a parity group with $N-1$ other lines (of the same offset) each from a different physical page. Intuitively, there is still some spatial locality/correlation among nearby pages, but the locality is perhaps not as strong as that between lines within a page. To quantify this locality, we use *group complete probability (GCP)*. GCP is the probability of dirty victim lines belonging to a complete group, i.e., all the lines belonging to the parity group of the victim are in cache. GCP measures how often we can generate the new parity without additional reads from memory.

One thing worth noting here is that other lines in the group may be dirty too, which implies the main memory's version of the line is different from the cache version. In that case, we have two choices: (a) treat the line as not in cache, which makes the group incomplete (and base on the number of lines present pick either RAIM 5a or 5b); and (b) write back the line so that both cache and memory versions are the same. While we will discuss details later in Sec. 4.3, some general observations can be made. When parity groups are formed from lines in different pages, the GCP is, at about 21%, understandably low. When lines from the same page are used to form group, GCP improves more than 3x to average around 63%.

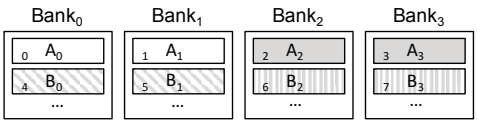
To make lines from the same page be part of a parity group, we need them to be in different channels (for errors to be more independent). We do this by offsetting each subsequent line by one channel and wrap around in a helix-like fashion (Fig. 3). For convenience of discussion we refer to consecutive lines in one of the strands a *frame*. For example, in Fig. 3, A_0 to A_3 form a frame (of 4 lines long).

As with line interleaving, in our mapping, neighboring lines in physical address space are spread across different channels, balancing load. But, importantly, unlike line interleaving, where *n* channels of space is essentially merged together to create one address space, our mapping preserves *n* frames allowed by *n* channels. This allows the operating system the flexibility to either expose all pages for software access or maintain some for parity purposes. The physical address space mapping only depends on frame size. If the frame size is set to *M* lines, then the first *M* lines in physical address space (0 to $M-1$) will be in frame 0, and line *M* will be the 0^{th} line of frame 1. Note that as long as the frame size is a multiple of an OS page size, there is no other constraints for our purpose and it can even be the entire capacity of a channel. However, if we set *M* to be a multiple of *N* (number of data lines in a parity group), the address calculation is simpler.

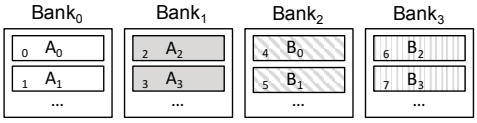
Given physical line number *pln* (physical address right-shifted to eliminate line offset), and frame size FS (the number of lines in a frame, which is both a multiple of the number of lines in a page and a multiple of *N*), we can express *pln* as frame ID fid and frame offset fo, then a line is mapped into channel $fid + fo\%N + 1$. The physical line number of other lines in the same protection is simply $\lfloor plnN \rfloor * N + i, i = 1, 2.., N-1$. The physical line number of the parity line is $pln - pln\%N + fid\%N + 1 + N - fid\%N + 1 \times FS$. Note that the seeming complexity of the formula can be misleading. For instance, the last part of the formula $(N - fid\%N + 1 \times FS)$ is only to set the address to a fixed frame where the parity lies. If *N* is, say, 4, then this part merely dictates setting the last two bits of the frame id to binary "11".

In summary, frame is a logical concept merely used to construct the mapping function. For simplicity, we choose a frame size to be both a multiple of page size, and the number of lines in a parity group. For example, in 3+1 configuration, assuming 4K page size, then a good frame size will be 12KB.[3]

Cache interleaving: In addition to memory interleaving, there is also the issue of cache bank interleaving. Conventionally, cache banks use line interleaving to avoid hot spot. With a typical interleaving, the member lines in a parity group will be distributed among N cache banks. Even if the group is complete, a single cache controller does not have the knowledge until it polls other cache banks. A more convenient configuration is to do parity group interleaving, keeping the entire parity group in one bank of the last level cache. Fig. 4 illustrates the difference. When both memory and cache interleaving are applied at the same time, we call it HGM (high GCP mapping).



(a) Conventional cache line interleaving



(b) Proposed parity group interleaving

Figure 4: Mapping of physical lines into last-level caches. This figures illustrates a system with 4 banks. Each small box represents a cache line, and the number on the left is the line number of the cache line. A_0 and A_1, A_2 and A_3, B_0 and B_1, and B_2 and B_3 form four parity groups.

Note that mapping lines in a parity group to the same cache bank does not necessarily reduce performance. The alternative (of cache interleaving) trades locality for bank-level parallelism. Also note that the serialized access is faithfully modeled in our simulation. We find this serialization to have little performance impact.

In summary, our RAIM-5 design posts some constraints on the address interleaving schemes. But the change is small and mostly affects the path of writebacks and is thus not on the critical path.

Other techniques: We briefly discuss a few other techniques explored to reduce overhead.

- Improving group completeness: Recall that when the whole parity group is present in cache, the writeback overhead is low. One way to boost such completeness for groups with dirty data is to prefetch missing lines. However, prefetching this way may not help cache hit rate. So we only prefetch when a dirty line is becoming the next in line to be evicted and there is only one line missing from the group. This technique is effective in baseline mapping, noticeably reducing traffic (by 22%). But with our proposed interleaving dramatically improving the group complete probability, the prefetch shows little impact and becomes unnecessary.

  Another way to boost group completeness we tried is to promote all member lines to the MRU position in their respective sets when one line is written to. This also has virtually no impact (improving by about 1%) with our proposed interleaving scheme.
- Writeback before losing group completeness: If evicting a clean cache line makes a group lose its completeness, then we can take the opportunity to write back any dirty data in the same group. This also has a very small impact.
- Reduce writebacks: We attempted to give dirty lines more chances to stay in cache without being evicted, for example, by making it twice as long to demote them. This does not work and increases traffic.

3.4 Putting It All Together

Fig. 5 illustrates how software applications can utilize RAIM-5's selective protection feature and how virtual and physical memory are managed by the OS. As can be seen, the actual memory management and virtual to physical mapping are still based on pages, thus existing MMU mechanisms remain largely unchanged.

User applications need to tell OS the protection requirement for different data blocks at the granularity of frame size. One way to implement this is to modify `malloc()` in user space and `mmap()` and `brk()` in the kernel. For example, when calling `malloc()`, the user also specifies the protection requirement. Upon requesting virtual memory from the kernel, `mmap()` or `brk()` is invoked with the proper protection flag. Another way is to simply introduce a new system call for setting protection requirements for different blocks.

When OS allocates physical memory, the allocation granularity is the size of a protection group, which consists of $N+1$ frames. For a protected protection group, there are N data frames and 1 parity frame; for an unprotected one, all $N+1$ frames are data frames. The entire group can be marked as protected or unprotected, thus all the pages in the same group have the same protection property. This information is recorded using an extra bit in the page table as well as TLB entries. The physical memory allocator is also modified so that it can allocate memory from the proper protection group. A portion of page table and TLB entries of the corresponding mapping are also illustrated in Fig. 5.

Finally, when gang writeback and high GCP mapping are enabled, lines are mapped in a helix fashion as illustrated in the right-most column of Fig. 5.

In summary, our RAIM-5 design requires only minor hardware modifications. And all such modifications are limited to the the LLC (L2 in our experimental setup) and memory controllers. Our optimizations (gang writeback and cache interleaving) only change where an LLC line is mapped and how to evict a dirty line. The rest of the system, especially, the interface with the L1 caches remains

[3] It is worth pointing out that when a frame is storing parity lines, its address range is not visible to any process and thus no cache lines within that address range will be requested. In other words, there are holes in the address space and they can make certain sets in the cache to be underutilized. We can choose to ignore this problem or to select a frame size to avoid the problem completely. If ignored, our empirical data suggest that this problem will cause the number of misses to increase a few percent on average. To completely eliminate this problem, we can add one more constraint to the frame size: that it is a multiple of the size of a cache way (in LLC).

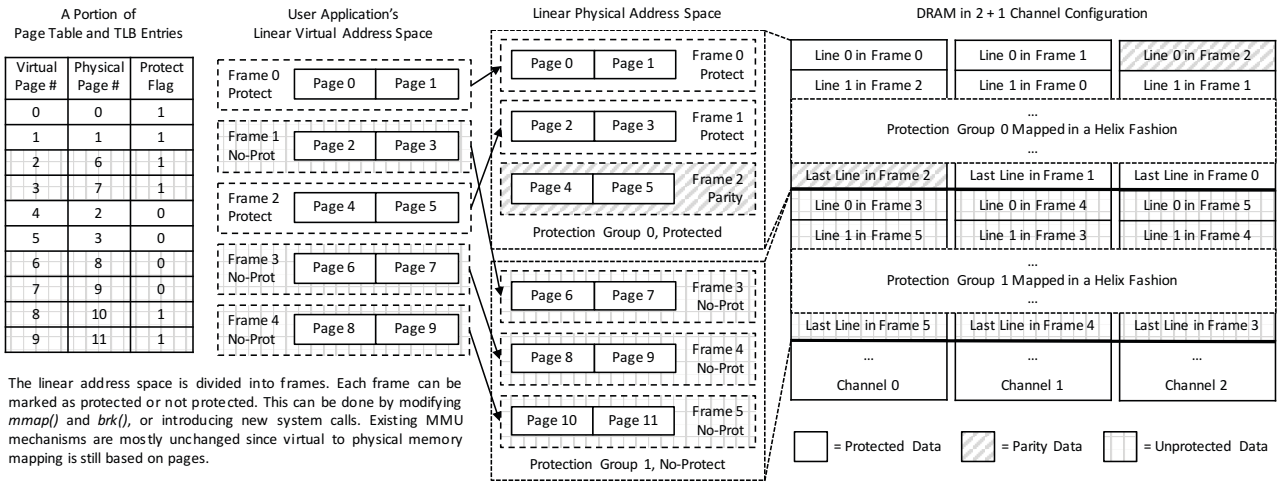


Figure 5: Memory management in RAIM-5. The figure illustrates how RAIM-5 manages virtual and physical memory, and how frames and protection groups are formed from pages. As can be seen, memory management is still based on pages. Note that in a protected group, the parity frame is not visible to any user process.

the same as the baseline system. For example, with or with our our RAIM-5 logic, when the LLC evicts a dirty line it would need to search L1s and request writebacks.

4 EXPERIMENTAL ANALYSIS

In this section, we first discuss experimental setup (Sec. 4.1), then discuss the overall effect of our proposed RAIM design (Sec. 3.2), and finally present detailed analyses (Sec. 4.3).

4.1 Experimental Methodology

We evaluate RAIM using gem5 [4] simulator with Ruby memory model. DRAM is modeled with the DRAM model embedded in gem5 [11]. The simulated OS uses 4KB page size. Table 2 shows the detailed simulation parameters. Our baseline machine has 5 data channels protected by ECC only and cache line size is 64B. All the results are normalized to this baseline unless otherwise specified. To minimize non-determinism, for each application, we either collect data between two global synchronization points, or run 1 billion useful instructions (excluding instructions in kernel mode and for spinning) across all cores. The average result of multiple runs is reported.

Our application mix represents both conventional and emerging workloads. We run selected PARSEC [3] applications using `simlarge` input sets. Since memory traffic is of primary interest in our experiments, a few applications with too little traffic in the simulation window[4] or with too much non-determinism across runs (`ferret`) are excluded. We also run popular graph applications from the CRONO benchmark suite [1], including depth-first search (`dfs`), community (`com`), page rank (`page`), and betweenness centrality (`bc`). The input is a synthetic graph with 327680 vertices and fixed 16 edges per vertex. This input size is much larger than the that of the L2 cache.

[4]These are `blackscholes`, `bodytrack`, `fluidanimate`, and `swaptions`. They generate less than 5MB of traffic in the window. The traffic of the rest of the applications ranges from 11MB to 700+ MB.

Table 2: Simulation Parameters

Cores	OoO, 2GHz, 16-core, 2D-mesh, MESI
Width	8-wide issue and commit
Issue queue	64-entry unified
Load-store queue	32-entry LD queue, 32-entry ST queue
Reorder buffer	192-entry
Physical registers	256-int, 256-fp
Branch predictor	2K local, 8K global, 4K BTB, 16 RAS
Private L1I&L1D	64KB each, 4-way, 64B line, 2-cycle
Distributed shared L2	4MB total, 16-way, 64B line, 16-cycle
DDR3-1600 DRAM	5-channel (4-data + 1-parity), 1GB/ch

4.2 Overall Effects

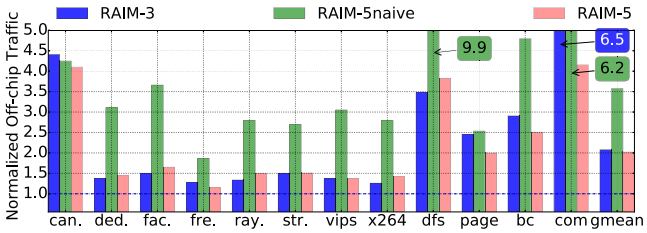
Our proposed RAIM-5 design is intended as a better alternative to RAIM-3 with numerous advantages: highly flexible selective protection and flexibility in cache line size and the number of memory channels. The tradeoff is that with a naive implementation (i.e., without the proposed optimizations), there will be significant additional costs associated with memory accesses. We therefore first answer the bottom-line question: can the proposed optimizations make the cost of our RAIM-5 architecture sufficiently low to justify the advantages gained. Fig. 6 focuses on the main cost – off-chip traffic.

Off-chip traffic: Note that both RAIM designs can have different configurations. In this first comparison, we take the RAIM-3 configuration used in the real world [30] and compare a RAIM-5 design with the same functionality. Note that RAIM-3 uses 256B cache line size for better execution time (more on that later), whereas baseline and RAIM-5 configurations use 64B cache-line size. In these comparisons, we use 4+1 channels for RAIM systems and compare the traffic to a system without RAIM and thus all 5 channels are for

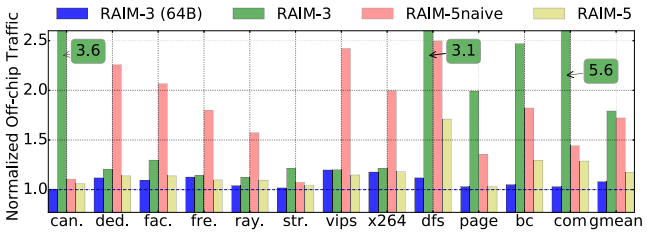
data accesses. Finally, in all our experiments, RAIM-5 is (conservatively) assumed to be protecting all memory, without any benefit from selectivity.

Fig. 6-a shows the normalized off-chip traffic. We see that by engaging RAIM-3, the traffic rises to about 2x on average. A naive implementation of RAIM-5, however, makes the traffic 3.5x that of the baseline on average. By applying the optimizations discussed, the traffic drops back to 2x. In other words, an optimized RAIM-5 system no longer has significant additional cost compared to RAIM-3.

Note that for a few applications (e.g., `com` and `dfs`) the overhead is still high after a large reduction. This is largely due to the applications' poor locality and irregular access patterns making fetching and caching the entire parity group a bad choice. We see that RAIM-3 also suffers from the same problem and has similarly high relative traffic.



(a) Default configuration



(b) Optimistic configuration

Figure 6: Off-chip traffic normalized to baseline. We count the total amount of data transferred between main memory and the CPU.

As mentioned already, there are a variety of ways one can configure a RAIM system. A particular configuration (which we call optimistic) worth noting is when we use the redundant channel only when the per-channel ECC mechanism detects, but can not correct an error. This is in contrast to the default configuration, where the parity channel is accessed for all memory requests and can thus detect certain errors that per-channel ECC fails to detect. This optimistic configuration does not have the same coverage as the default configuration but can still cover most errors including chip and channel failures. Under error-free conditions, the overhead of the optimistic configuration is only limited to that from writebacks. As shown in Fig. 6-b, compared to baseline, the normalized memory traffic would only be 1.17x in the RAIM-5 design, down from 1.72x in a naive implementation. We note that an optimistic version of RAIM-3 also sees a significantly lower traffic overhead: 1.79x and 1.08x for 256B and 64B cache-line size, respectively.

Again, the overhead shown here is assuming RAIM-protecting the entire memory. The cost of selectively protecting portions of memory is proportionally lower for RAIM-5. When no memory is protected, the difference in memory traffic compared to a baseline system (which has different mappings) is negligible. In other words, RAIM-5 is cost-proportional.

Memory energy costs: Memory traffic statistics only tell part of the story of RAIM system overhead. The energy consumption of memory accesses is another part. Energy consumption contains a fixed portion due to refreshing and the dynamic portion depends on on how these accesses are spread across channels and their hit rates on row buffer. RAIM-5 generally has lower overhead in memory energy compared to RAIM-3. As shown in Fig. 7, RAIM-3 increases memory energy by an average of 54% (with a range of 2% to 173%). RAIM-5 configurations keep the energy consumption overhead at a much lower 28% (with a range of 5% to 127%). This is less than half of the 61% overhead of a naive implementation.

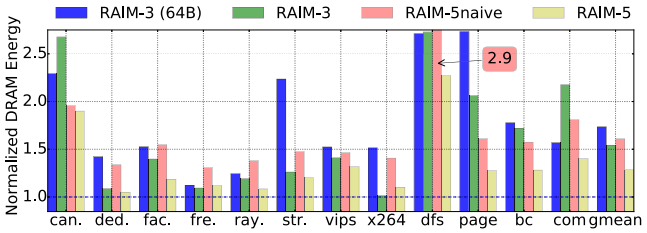


Figure 7: Total memory energy consumption normalized to baseline.

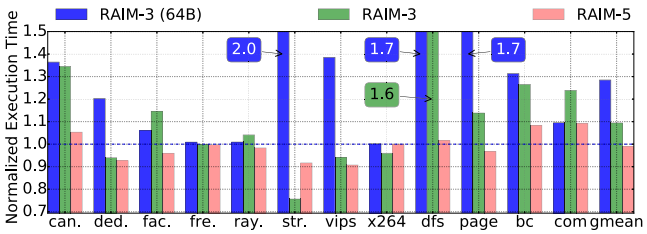


Figure 8: Execution time normalized to baseline.

Execution time impact: Finally, we show the impact of different RAIM designs on the execution time in Fig. 8. Like previous results, the baseline for comparison uses all $N+1$ channels for regular memory accesses. Hence, it enjoys more effective memory bandwidth. We can see that the performance difference is largely muted across most applications for RAIM-5 configurations. The small differences are largely because RAIM-5 caches more lines (the entire parity group) on a miss. This can improve performance in some applications and hurt in others. Overall, the average impact is negligible.

RAIM-3, on the other hand, presents an unfortunate design tradeoff that manifests more clearly in execution time. RAIM-3 spreads a cache line over multiple memory channels. A small cache line

size reduces burst length and multiplies overhead. Hence RAIM-3 designs use long cache lines at the LLC. For applications with poor spatial locality, however, this choice not only wastes cache space, but also memory bandwidth. `dfs` is one clear example where both long and short cache lines are equally poor choices for RAIM-3. In contrast, RAIM-5 removes the limitation on both the cache line size and the number of memory channels.

Recap: In summary, the proposed selective RAIM-5 design allows protection cost proportionality and flexibility in baseline configuration in terms of cache line size and the number of memory channels available. In exchange for these benefits, RAIM-5 pays with the cost of extra memory traffic, which can be significant in a naive implementation. With the proposed optimizations, this cost is significantly mitigated:

(1) Traffic overhead reduces from 250% to 100%, roughly inline with that of RAIM-3. In the case of an optimistic configuration, the memory traffic overhead drops from 72% in a naive implementation to 17% in the optimized design.
(2) Energy overhead of memory reduces from 61% in a naive design to 28% with optimizations. This compares to 54% in a RAIM-3 design.
(3) On average, the optimized RAIM-5 architecture has a negligible impact on the execution time, compared to an average of about 10% slowdown induced by RAIM-3.

Overall, we believe these statistics suggest that RAIM-5 is a compelling alternative to RAIM-3.

4.3 Detailed Analysis

We now dive in some of the details to better understand the design and the trade-offs.

Different number of memory channels: An important benefit of the proposed design is that it offers high on-demand protection without imposing constraints on the number of channels. RAIM 3, on the other hand, is much less practical in other configurations. Fig. 9 shows the cost in traffic and DRAM energy of RAIM-5 having different number of memory channels. Each bar is normalized to its corresponding baseline ($n+1$ channel RAIM-5 is normalized to $n+1$ channel baseline). Recall that if an optimistic configuration of RAIM is used, there will be less overhead. This result is shown using darker colors for each bar.

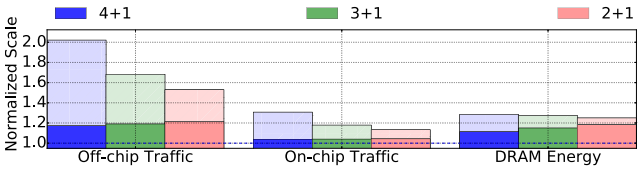


Figure 9: Effect of having different number of channels, all normalized to its corresponding baseline. Each bar represents the geometric mean across all applications. The optimistic configuration incurs much less overhead, which is shown as the portion in darker colors.

As can be seen, with fewer channels, the overhead of accessing the entire parity group becomes smaller. The tradeoff is that storage overhead is higher with fewer channels to form RAIM. In the case of RAIM-5, this storage overhead applies only to the portion of the memory selected for protection.

Gang writeback and mapping: Fig. 10 shows the effect of the optimizations in more detail. Recall a RAIM-5 writeback can have two flavors (which we termed 5a and 5b). A naive RAIM-5 design picks the better of the two depending on the situation for each individual writeback. Thus the naive RAIM-5 design is slightly better than either 5a or 5b in terms of off-chip traffic. However, the overhead is still quite high (total traffic is 2.6x or 72% more than the underlying baseline system for default and optimistic configurations respectively). We see that the high GCP mapping (HGM) and gang writeback (GW) bring the overhead down significantly (to 102% and 17% for default and optimistic configurations respectively). Moreover, they are synergistic. Table 3 shows this synergy more clearly. Note that the page size is 4KB.

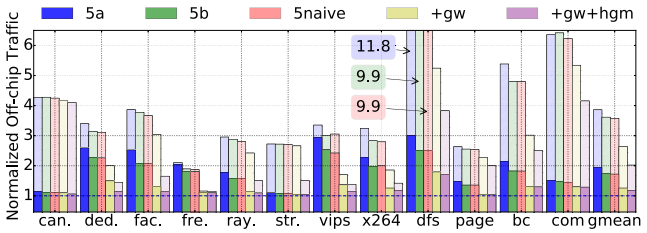


Figure 10: Off-chip traffic comparing, naive implementation of RAIM-5 (`5naive`), GW (`+gw`), together (`+gw+hgm`), as well as RAIM-5a (`5a`) and RAIM-5b (`5b`). The lighter colors shows the default configuration and the darker colors the optimistic configuration.

Table 3: Group complete probability (GCP) under different configurations: original system (Base), with gang write back (GW), and adding high GCP mapping (HGM).

Application	Base	GW only	GW+HGM
can.	<0.1%	<0.1%	12.7% (>400x)
ded.	13.1%	26.1%	80.3% (6.1x)
fre.	24.6%	79.5%	83.9% (3.4x)
fac.	14.2%	32.8%	79.3% (5.6x)
ray.	15.1%	40.7%	69.3% (4.6x)
str.	10.2%	14.7%	41.1% (4.0x)
vips	15.4%	38.1%	84.8% (5.5x)
x264	19.6%	48.4%	76.1% (3.9x)
page	27.3%	60.3%	97.9% (3.6x)
bc	45.9%	84.7%	99.6% (2.2x)
com	5.7%	5.8%	6.9% (1.2x)
dfs	52.6%	95.8%	97.0% (1.8x)
Average	20.9%	45.5%	63.3% (3.2x)

The table shows GCP under different configurations. We can see that the GCP is the highest when both techniques are combined. The improvement ranges widely and has a geometric mean of more than 3x. Indeed, when the group is complete, we have to write back all

dirty lines. Otherwise the newly generated checkpoint from the dirty data will not be consistent with memory data. But fortunately it is also a good time to write back: if we do not take the opportunity to gang writeback, the eviction will make the group incomplete, making other writebacks more expensive. We find that on average only 7.3% of writebacks are premature, that is, the line becomes dirty again after the gang writeback.

Access statistics: Fig. 11 shows the number of read and write accesses from the memory controllers and the on-chip traffic in different schemes. The results reveal an inherent challenge in the RAIM-3 design. When using a RAIM-3 design that spreads a single cache line across multiple channels, every last-level cache miss involves multiple memory transactions on separate channels. This is part of the reason that energy consumption is high for RAIM-3. To amortize this cost, actual designs use much a longer cache line size (e.g., 256B), which has its own drawback. For instance, in our setup, the on-chip traffic in RAIM-3 (112%) is significantly higher than baseline. In contrast, on-chip traffic in our proposed RAIM-5 design (31%) is much less.

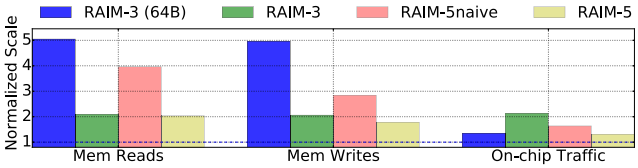
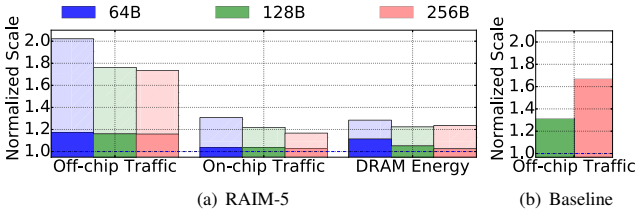


Figure 11: Total number of memory reads, writes, and on-chip traffic, all normalized to baseline. Each bar represents the geometric mean across all applications. Traffic is measured in bytes.

Different cache line sizes: Unlike RAIM-3, which essentially places a constraint on the cache line size, RAIM-5 is rather insensitive to the choice of line size. As we can see in Fig. 12-a, the overhead appears lower with longer cache lines. In reality, smaller cache line sizes make gang-writeback somewhat more efficient due to better locality, but the difference is small. The biggest factor in the difference of relative off-chip traffic is the traffic in the baseline system without RAIM. As we increase the cache line size, the baseline system starts to be more inefficient (more traffic and DRAM energy) as we can see in Fig. 12-b. This larger base makes the overhead of RAIM-5 appear smaller in relation.

Different page and frame sizes: Modern processors usually support multiple page sizes, which range from the conventional 4KB to several gigabytes. Page size does not directly affect our RAIM-5 design since page-based memory management is not changed in our design. However, page size affect frame size. Frame size has to be a multiple of page size and a multiple of a cache way. It is possible that a large page size makes frame size also large. Because the granularity of selective protection is a frame, a larger frame can make the granularity coarser. Other than this, a large frame size does not have significant impact on our RAIM-5 system.

With different page sizes, the applications' performance will be affected. It is hard to isolate the effect of page size itself and the effect of RAIM-5. So in this set of experiments, based on the above discussion, we keep the page size constant as 4KB and change the frame size from 64KB to 1GB. As is shown in Fig. 13, our RAIM-5 with different frame sizes have a consistent behavior.



(a) RAIM-5 (b) Baseline

Figure 12: Effect of different cache line sizes on traffic and memory energy of RAIM-5. (a) Each bar represents the geometric mean across all applications normalized to its corresponding baseline (i.e., the 256B RAIM-5 is normalized to a 256B baseline). Darker color portion of the bar indicates the overhead when running RAIM-5 in optimistic mode. (b) Off-chip traffic in a baseline system with different line sizes all relative to that of 64B baseline.

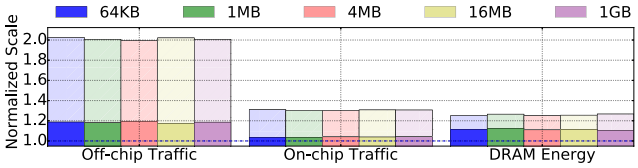
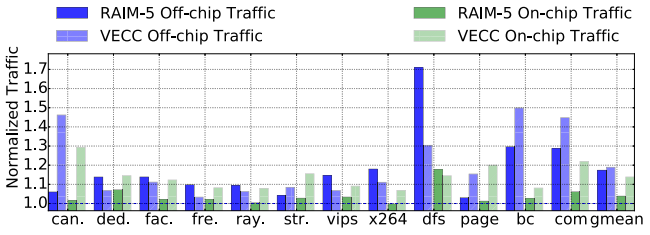


Figure 13: Effect of different frame sizes.

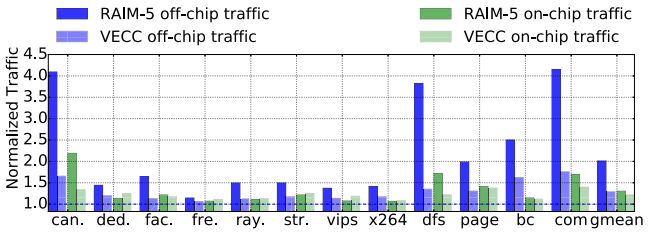
RAIM vs. Virtualized Ecc: Our proposed RAIM-5 design and Virtualized ECC (VECC) are two different proposals supporting selective protection but with somewhat different emphases. RAIM-5 was intended as a more flexible, and cost-proportional alternative to RAIM-3, which is intended for high-end systems. For a higher overhead, RAIMs offer strong protection such as against channel failures. In contrast, Virtualized ECC only tolerates chip failures. Both designs can also be adapted to offer ECC protections to systems using non-ECC DIMMs. However, this was not the intent of the RAIM-5 design.

In the following comparison, we focus on traffic costs of RAIM-5 vs. Virtualized ECC in two scenarios. In the first scenario, both designs use the additional storage to augment regular per-channel ECC. In the second scenario, the additional storage provides the only form of ECC. Note that the original Virtualized ECC design [49] was mainly for a single-core system and did not discuss address mapping in multi-banked last-level caches. We extended the original design to make it support multi-banked last-level caches. We model x8 chips for both RAIM-5 and Virtualized ECC for comparison.

In Fig. 14-a, we can see that RAIM-5 incurs similar overheads in off-chip traffic as Virtualized ECC (17% vs. 19% on average), but less in on-chip traffic (4% vs. 14% on average). The two configurations also have similar performance, with Virtualized ECC being about 1% slower. In the second scenario, we see that RAIM-5 has a much higher overhead in off-chip traffic (102% vs. 29% on



(a) Scenario 1



(b) Scenario 2

Figure 14: Off-chip and on-chip traffic in two scenarios of using RAIM-5 vs. Virtualized ECC.

average), a somewhat higher overhead in on-chip traffic (31% vs. 22% on average); though RAIM-5 has a performance advantage of 5%. Again, this scenario is not the intent of the RAIM-5 design.

Generally speaking, we see Virtualized ECC as a good mechanism to add to mainstream systems to offer a stronger protection, especially to systems using non-ECC DIMMs. On the other hand, RAIM-5 is a compelling alternative to RAIM-3 architectures found in high-end systems.

5 CASE STUDY OF SELECTIVE PROTECTION

With our proposed system and proper OS support, there is already flexibility for end users in selective protection. For instance, a user can enable protection for important runs and only pay the cost then. However, an additional level of flexibility is to protect different data differently. One of the motivating categories of such selective protection is large scale scientific applications. Scientific computation is a pillar for modern scientific exploration. On the one hand, due to the size of these applications, memory errors are a statistical certainty, and the consequence can be severe.[5] On the other hand, without selective protection, traditional counter measures can be an overkill as many scientific codes are large *because* they are modeling a complex system and errors in one component need not necessarily compromise system integrity. In this paper, we use a particular scientific computation application – a particle-in-cell (PIC) simulator – for a limited case study of such selective protection.

[5]There is a common misconception that scientific computations do not need the same level of protection as business operations do. The reason scientific computing platforms today often provide very limited reliability measures is largely an economic one. Scientists would be just as happy as bankers to not have to worry about computational errors.

5.1 Introducing OSIRIS

A PIC code is a generic simulator that relies on low-level modeling of particles in a physical space. It is a first-principle simulator that is often used to validate other derived models of simulation. The code structure itself is generic enough to allow modeling of a whole range of phenomena from astrophysics to plasma research. We use this code to model Landau Damping in this study. We chose this study for three reasons. First, Landau Damping is a prime example of the use of computing in science. The phenomenon is first discussed some 70 years ago [22] and remains the "single most famous mystery" of classical plasma [32]. The phenomenon defies simple theoretical characterization and simulation is a prime, if not the only way to accurately model it. Second, when studying a large number of particles, there is typically no easy way to summarize the outcome of a simulation. In the case of simulating Landau damping, we can actually reduce the whole simulation to a single scalar value describing the damping rate. This serves as a convenient first-order characterization of how much impact errors have on the computation. Finally, we are familiar with the prime academic code for PIC simulation called OSIRIS, which is one of the most widely used PIC code and a large player on the Department of Energy's most powerful machines.

5.2 The Observation

We first relay an interesting anecdote. A production run of OSIRIS can easily recruit 10,000 computing nodes and runs for weeks. So a typical production simulation does not lend itself to simple verification from reruns. One such run produced results that are so unlikely to be real that physicists start to pore over checkpoint dumps and indeed found evidence that computer error is to blame (Fig. 15).

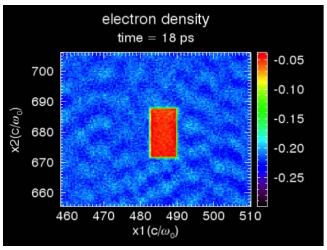


Figure 15: Illustration for the error caught in diagnostic data. The visualization shows a rectangle region with abnormal electron density that clearly has roots in data structure error uncaught by the code.

Intuitively, since the code models an enormous number of individual particles, errors in any one particle is incapable of altering the overall behavior of the system. Since memory errors are still rare, certain aspects of the particle data can be protected against memory error in a variety of alternative ways including having no protection at all. In other words, some data structures inherently offer graceful degradation. Indeed, while uncorrectable errors will almost always result in the termination of an application, in some cases, the best thing to do may be simply to ignore it and carry on. In the case of OSIRIS, particle momentum data, a single data structure (array), is one such data example. If a particle has "gone rogue" and acquired

an erroneous momentum, the error will generally be too small to affect the plasma as a whole.

5.3 Catastrophe Recovery

Although the majority of data can naturally tolerate some inaccuracy, as it turns out, this does not mean the data is impervious to any errors. For example, an error occurring to the exponent bits may produce physically meaningless values or cause subsequent computation to overflow and produce NaN or Inf, which in turn may cause the program to crash. However, upon such a catastrophe, the program does not necessarily have to crash. In our case study, we make simple modifications to ignore the catastrophe and keep on running. First, we embed a signal handler into the program. This code handles the floating-point exceptions caused by NaN and Inf numbers, directly preventing the crash. However, addressing the symptom alone is not enough. So the second thing we do is to clean up obvious corruptions. This can help reduce the chance of future crashes, and thus prevent the signal handler from getting called too frequently. Upon being triggered, the handler also scans the array for wrong (e.g., NaN) or unreasonably large values and replace them with a proper random value.[6] We also modify the code to prevent the errors from producing wrong index values for array accesses.

The impact on non-uniform protection cost is significant. In this example, in a production run, the size of momentum data can reach beyond 90% of all data footprint. We show in Sec. 5.4 that our intuition is born out by experiments. In this particular case, both the physical meaning and identification of the data structure are rather straightforward.

5.4 Error Injection Study

In this set of experiments, we inject single bit errors to different arrays in OSIRIS under different configurations comparing no protection with selective protection, where the hardware only protects the subset of crucial data. We label the momentum data as robust (and therefore they do not need RAIM protection) but enable the software catastrophe recovery. The rest of the data (including code, stack, some data arrays, loop counters, pointers, etc.) are considered "vulnerable". In Fig. 16, we show the crashing rate and error of Landau damping rate of the survived runs under different FIT rates. The error injections are intentionally raised to levels higher than observed DRAM FIT rates [41].

We see that without hardware protection, using only our software catastrophe recovery, the robust data can sustain much more memory errors than the vulnerable data. For error occurring in vulnerable data, as the error rate increases beyond 10^4 FIT, we start to see non-trivial rate of crashing. For those runs that did not crash, the end result is essentially useless. On the other hand, when error occurs to the robust data, no crash occurred even if we increase the rate by six orders of magnitude. In all cases, the results are largely tolerable and well within the range of randomness induced variation.

Finally, we run OSIRIS with selective RAIM-5 protection. Compared to RAIM-3, we achieve about 80% savings in storage overhead. Execution time and DRAM energy overhead become essentially negligible.

[6]This is straightforward as there is only a single type of data (momentum) to scan and their values are known to follow Maxwellian distribution throughout time.

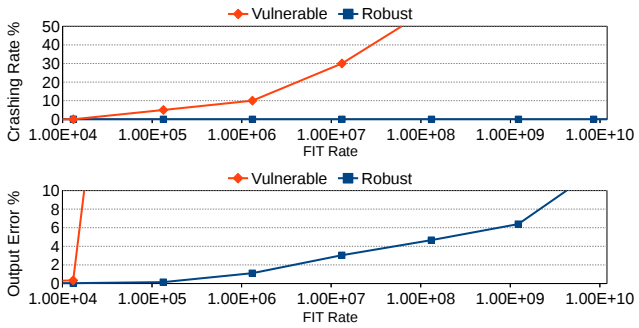


Figure 16: Percentage of crashing rate and Landau damping rate error.

6 CONCLUSIONS

Errors in the main memory are a serious source of overall system reliability concern. While more efficient error-correcting codes are constantly being proposed, another source of efficiency gain can be obtained by recognizing the vast differences in applications' protection needs. This is important as we are increasingly consolidating diverse workloads to the cloud environments. In this paper, we proposed an alternative RAIM-based design that offers the strongest memory protection available in commercial products. Our design not only provides better implementation flexibility in terms of number of memory channels needed and appropriate cache line sizes, it also easily provides selective protection. Additionally, we presented design optimizations to lower its overhead, mainly in memory traffic.

The result is a design that produces the highest error protection when needed under a reasonable overhead. In a 5-channel implementation, the average traffic overhead is 101% over the underlying baseline architecture, compared to 108% in a state-of-the-art RAIM-3 design. The memory energy overhead is 28% for RAIM-5 compared to 54% for RAIM-3. The traffic overhead results in negligible impact on execution speed for RAIM-5. RAIM-3, on the other hand, incurs about 10% execution speed penalty. The behavior of RAIM-5 is rather consistent across a variety of parameters such as cache line size and page size. A case study with a production quality scientific workload shows that a user of the code can easily pick out a large portion of the data structure to be excluded from system-level memory protection, which drastically lowers the overheads without impacting system reliability. Overall, we believe the proposed optimized RAIM-5 design is a compelling mechanism for high-end systems already employing RAIM-3 and for low-end systems as an additional cost-proportional memory protection mechanism.

REFERENCES

[1] Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. 2015. Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *Proceedings of IEEE International Symposium on Workload Characterization*. 44–55.

[2] Amazon 2008. Amazon S3 Availability Event. (2008). http://status.aws.amazon.com/s3-20080720.html.

[3] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Computer Architecture News* 39, 2 (Aug. 2011), 1–7.

[5] Shekhar Borkar. 2005. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro* 25, 6 (Nov.–Dec. 2005), 10–16.

[6] Long Chen and Zhao Zhang. 2014. MemGuard: A low cost and energy efficient design to support and enhance memory system reliability. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. 49–60.

[7] Jason Cong and Karthik Gururaj. 2011. Assuring Application-level Correctness Against Soft Errors. In *Proceedings of the International Conference on Computer-Aided Design*. 150–157.

[8] Cristian Constantinescu. 2002. Impact of Deep Submicron Technology on Dependability of VLSI Circuits. In *22nd International Conference on Dependable Systems and Networks*. Bethesda, MD, 205–209.

[9] Timothy J Dell. 1997. *A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory*. IBM Microelectronics Division.

[10] Nosayba El-Sayed, Ioan A Stefanovici, George Amvrosiadis, Andy A Hwang, and Bianca Schroeder. 2012. Temperature Management in Data Centers: Why Some (Might) Like It Hot. In *ACM SIGMETRICS*. London, UK.

[11] Andreas Hansson, Neha Agarwal, Aasheesh Kolli, Thomas Wenisch, and Aniruddha N Udipi. 2014. Simulating DRAM controllers for future system architecture exploration. In *Performance Analysis of Systems and Software, 2014 IEEE International Symposium on*. 201–210.

[12] HP 2008. HP Advanced Memory Protection technologies. (2008). ftp://ftp.hp.com/pub/c-products/servers/options/c00256943.pdf.

[13] Xun Jian, John Sartori, Henry Duwe, and Rakesh Kumar. 2013. High Performance, Energy Efficient Chipkill Correct Memory with Multidimensional Parity. *Computer Architecture Letters* 12 (Dec. 2013), 39–42.

[14] Xun Jian, Vilas Sridharan, and Rakesh Kumar. 2016. Parity Helix: Efficient protection for single-dimensional faults in multi-dimensional memory systems. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 555–567.

[15] Manolis Kaliorakis, Sotiris Tselonis, Athanasios Chatzidimitriou, Nikos Foutris, and Dimitris Gizopoulos. 2015. Differential Fault Injection on Microarchitectural Simulators. In *2015 IEEE International Symposium onWorkload Characterization*. 172–182.

[16] Ron Kalla, Balaram Sinharoy, William J Starke, and Michael Floyd. 2012. *POWER7 System RAS: Key Aspects of Power Systems Reliability, Availability, and Serviceability*. Technical Report. IBM Server and Technology Group.

[17] Samira Khan, Donghyuk Lee, Yoongu Kim, Alaa R Alameldeen, Chris Wilkerson, and Onur Mutlu. 2014. The efficacy of error mitigation techniques for DRAM retention failures: a comparative experimental study. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 42. 519–532.

[18] Daya Shanker Khudia and Scott Mahlke. 2014. Harnessing soft computations for low-budget fault tolerance. In *Proceedings of the International Symposium on Microarchitecture*. 319–330.

[19] Jungrae Kim, Michael Sullivan, and Mattan Erez. 2015. Bamboo ECC: Strong, safe, and flexible codes for reliable computer memory. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 101–112.

[20] Soontae Kim. 2006. Area-Efficient Error Protection for Caches. In *Proceedings of the Design, Automation and Test in Europe*. 1282–1287.

[21] Seongwoo Kim and Arun K Somani. 1999. Area Efficient Architectures for Information Integrity in Cache Memories. In *Proceedings of the International Symposium on Computer Architecture*. 246–255.

[22] Lev Davidovich Landau. 1946. On the vibration of the electronic plasma. *J. Phys. USSR* 16, 574 (1946), 25–34.

[23] Donghyuk Lee, Yoongu Kim, Gennady Pekhimenko, Samira Khan, Vivek Seshadri, Kevin Chang, and Onur Mutlu. 2015. Adaptive-latency DRAM: Optimizing DRAM timing for the common-case. In *Proceedings of the International Symposium on High Performance Computer Architecture*. 489–501.

[24] Xin Li, Michael C Huang, Kai Shen, and Lingkun Chu. 2010. A Realistic Evaluation of Memory Hardware Errors and Software System Susceptibility. In *Proceedings of the USENIX Annual Technical Conference*.

[25] Xin Li, Kai Shen, Michael C Huang, and Lingkun Chu. 2007. A Memory Soft Error Measurement on Production Systems. In *Proceedings of the USENIX Annual Technical Conference*.

[26] Xuanhua Li and Donald Yeung. 2007. Application-Level Correctness and its Impact on Fault Tolerance. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 181–192.

[27] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2015. Clover: Compiler directed lightweight soft error resilience. In *ACM Sigplan Notices*, Vol. 50. 2.

[28] Yixin Luo, Sriram Govindan, Bikash Sharma, Mark Santaniello, Justin Meza, Aman Kansal, Jie Liu, Badriddine Khessib, Kushagra Vaid, and Onur Mutlu. 2014. Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 467–478.

[29] Timothy C May and Murray H Woods. 1979. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices* 26, 1 (1979), 2–9.

[30] Patrick J Meaney, Luis Alfonso Lastras-Montaño, Vesselina K Papazova, Eldee Stephens, JS Johnson, Luiz C Alves, James A O'Connor, and William J Clarke. 2012. IBM zEnterprise redudandant array of independent memory subsystem. *IBM Journal of Research and Development* 56, 1/2 (2012), 4:1–4:11.

[31] Mojtaba Mehrara and Todd Austin. 2008. Exploiting Selective Placement for Low-cost Memory Protection. *ACM Transactions on Architecture and Code Optimization* 5, 3 (Nov. 2008), 14:1–14:24.

[32] Clément Mouhot and Cédric Villani. 2010. Landau damping. *J. Math. Phys.* 51, 1 (2010), 015204.

[33] Brendan Murphy. 2004. Automating software failure reporting. *ACM Queue* 2, 8 (Nov. 2004), 42–48.

[34] Prashant J Nair, Dae-Hyun Kim, and Moinuddin K Qureshi. 2013. ArchShield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 72–83.

[35] Eugene Normand. 1996. Single Event Upset at Ground Level. *IEEE Transactions on Nuclear Science* 43, 6 (1996), 2742–2750.

[36] Timothy J O'Gorman, John M Ross, Allen H Taber, James F Ziegler, Hans P Muhlfeld, Charles J Montrose, Huntington W Curtis, and James L Walsh. 1996. Field testing for cosmic ray soft errors in semiconductor memories. *IBM Journal of Research and Development* 40, 1 (1996), 41–50.

[37] David J Palframan, Nam Sung Kim, and Mikko H Lipasti. 2015. COP: to compress and protect main memory. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 682–693.

[38] Moinuddin K Qureshi, Dae-Hyun Kim, Samira Khan, Prashant J Nair, and Onur Mutlu. 2015. AVATAR: A variable-retention-time (VRT) aware refresh for DRAM systems. In *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 427–437.

[39] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I August, and Shubhendu S Mukherjee. 2005. Software-Controlled Fault Tolerance. *ACM Transactions on Architecture and Code Optimization* 2, 3 (Dec. 2005), 366–396.

[40] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. 2009. DRAM errors in the wild: a large-scale field study. In *ACM SIGMETRICS*. Seattle, WA, 193–204.

[41] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. 2015. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. 297–310.

[42] Sun Microsystems 2000. Sun Microsystems server memory failures. (2000). http://www.forbes.com/global/2000/1113/0323026a_print.html.

[43] Ayswarya Sundaram, Ameen Aakel, Derek Lockhart, Darshan Thaker, and Diana Franklin. 2008. Efficient Fault Tolerance in Multi-media Applications through Selective instruction Replication. In *Proceedings of the 2008 Workshop on Radiation Effects and Fault Tolerance in Nanometer Technologies*. 339–346.

[44] Dong Tang, Peter Carruthers, Zuheir Totari, and Michael W Shapiro. 2006. Assessment of the Effect of Memory Page Retirement on System RAS Against Hardware Faults. In *International Conference on Dependable Systems and Networks*. Philadelphia, PA, 365–370.

[45] Darshan D Thaker, Diana Franklin, John Oliver, Susmit Biswas, Derek Lockhart, Tzvetan Metodi, and Frederic T Chong. 2006. Characterization of Error-Tolerant Applications when Protecting Control Data. In *Proceedings of IEEE International Symposium on Workload Characterization*.

[46] Aniruddha N Udipi, Naveen Muralimanohar, Rajeev Balsubramonian, Al Davis, and Norman P Jouppi. 2012. LOT-ECC: Localized and Tiered Reliability Mechanisms for Commodity Memory Systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*. 285–296.

[47] Aniruddha N Udipi, Naveen Muralimanohar, Niladrish Chatterjee, Rajeev Balasubramonian, Al Davis, and Norman P Jouppi. 2010. Rethinking DRAM Design and Organization for Energy-constrained Multi-cores. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*. 175–186.

[48] Gary A Van Huben, KD Lamb, R Brett Tremaine, BE Aleman, SM Rubow, SH Rider, Warren E Maule, and Michael E Wazlowski. 2012. Sever-class DDR3 SDRAM memory buffer chip. *IBM Journal of Research and Development* 56, 1/2 (2012), 3:1–3:11.

[49] Doe Hyun Yoon and Mattan Erez. 2010. Virtualized and Flexible ECC for Main Memory. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*. 397–408.

[50] Wei Zhang. 2005. Replication Cache: A Small Fully Associative Cache to Improve Data Cache Reliability. *IEEE Trans. Comput.* 54, 12 (Dec. 2005), 1547–1555.

[51] James F Ziegler. 1996. Terrestrial cosmic rays. *IBM Journal of Research and Development* 40, 1 (1996), 19–39.

[52] James F Ziegler, Huntington W Curtis, Hans P Muhlfeld, Charles J Montrose, B Chin, Michael Nicewicz, CA Russell, Wen Y Wang, Leo B Freeman, P Hosier, and others. 1996. IBM Experiments in Soft Fails in Computer Electronics (1978-1994). *IBM Journal of Research and Development* 40, 1 (Jan. 1996), 3–18.

[53] James F Ziegler and WA Lanford. 1979. Effect of Cosmic Rays on Computer Memories. *Science* 206, 16 (Nov. 1979), 776–788.

[54] James F Ziegler, Hans P Muhlfeld, Charles J Montrose, Huntington W Curtis, Timothy J O'Gorman, and John M Ross. 1996. Accelerated testing for cosmic soft-error rate. *IBM Journal of Research and Development* 40, 1 (1996), 51–72.

[55] James F Ziegler, Martin E Nelson, James Dean Shell, R Jerry Peterson, Carl J Gelderloos, Hans P Muhlfeld, and Charles J Montrose. 1998. Cosmic Ray Soft Error Rates of 16-Mb DRAM Memory Chips. *IEEE Journal of Solid-State Circuits* 33, 2 (Feb. 1998), 246–252.