Using Managed Runtime Systems to Tolerate Holes in Wearable Memories

Tiejun Gao† Karin Strauss‡ Stephen M. Blackburn† Kathryn S. McKinley‡ Doug Burger‡ James Larus‡

†Australian National University
{tiejun.gao,steve.blackburn}@anu.edu.au

‡Microsoft Research
{kstrauss,mckinley,dburger,larus}@microsoft.com

Abstract

New memory technologies, such as phase-change memory (PCM), promise denser and cheaper main memory, and are expected to displace DRAM. However, many of them experience permanent failures far more quickly than DRAM. DRAM mechanisms that handle permanent failures rely on very low failure rates and, if directly applied to PCM, are extremely inefficient: Discarding a page when the first line fails wastes 98% of the memory.

This paper proposes low complexity cooperative software and hardware that handle failure rates as high as 50%. Our approach makes error handling transparent to the application by using the memory abstraction offered by managed languages. Once hardware error correction for a memory line is exhausted, rather than discarding the entire page, the hardware communicates the failed line to a failure-aware OS and runtime. The runtime ensures memory allocations never use failed lines and moves data when lines fail during program execution. This paper describes minimal extensions to an Immix mark-region garbage collector, which correctly utilizes pages with failed physical lines by skipping over failures. This paper also proposes hardware support that clusters failed lines at one end of a memory region to reduce fragmentation and improve performance under failures. Contrary to accepted hardware wisdom that advocates for wear-leveling, we show that with software support non-uniform failures delay the impact of memory failure. Together, these mechanisms incur no performance overhead when there are no failures and at failure levels of 10% to 50% suffer only an average overhead of 4% and 12%, respectively. These results indicate that hardware and software cooperation can greatly extend the life of wearable memories.

Categories and Subject Descriptors D. Software [*D.3 Programming Languages*]: D.3.4 Processors, Memory management (garbage collection); B. Hardware [*B.8 Performance and Reliability*]: B.8.1 Reliability, Testing, and Fault-Tolerance

General Terms Reliability

Keywords Failure tolerance, memory management, phase-change memory

1. Introduction

The semiconductor industry continues to exploit Moore's Law by scaling down lithographic feature sizes to improve chip density. However, due to increased process variation and smaller features, scaling charge-based memories such as DRAM is becoming increasingly difficult [12]. Errors increase as smaller features represent a bit with less charge. Process variation makes scaling even more challenging because different memory cells operate with a wider range of charges.

Memory manufacturers are delivering alternatives to charge-based memories, such as resistive and magneto-resistive memories. Among these, memristors are under advanced development and phase-change memory (PCM) is in production [15]. PCM's storage principle is based on physical atomic arrangements of the materials that constitute it, rather than charges it holds, resulting in more stable storage at smaller feature sizes. The result is better scaling for main memory storage, which will keep improving memory density and cost in smaller feature sizes. Moreover, PCM is non-volatile, which presents opportunities for new software designs [7] and lower power operation.

However, PCM comes with its own idiosyncrasies. Stable memory cells require more energy to write and fail much quicker compared to DRAM, as writes change the material's physical configuration and cause it to wear out. We call memories vulnerable to this issue *wearable memories*. This paper frames our solution as targeting PCM, but it is applicable to any main memory technology that suffers failures.

Current DRAM approaches assume failures occur extremely rarely. When one line in DRAM fails, software remaps its virtual page to another working physical page and discards the entire failed physical page [9]. Even recent work on error correction hardware tailored to wearable memories [16, 22, 23, 25] assumes that once the finite error correction resources are exhausted, the entire page, if not the entire memory, fails. Assuming only pages fail, 4 KB pages and 64 B lines, this solution wastes 98% of working memory. In other words, only 2% of lines need fail and the entire wearable memory becomes unusable.

This paper proposes a software/hardware cooperative mechanism to extend the life of wearable memories. We propose software mechanisms that handle failures at a fine granularity, e.g., a cache line, in which the OS and managed runtime extend the lifetime of wearable memory, with graceful performance degradation as the number of failures (*holes*) increases.

The first line of defense for failures is of course the hardware error correction of the memory system. When the number of failed bits exceeds the capacity of the error correction hardware, the memory communicates this failure to the OS. The OS keeps track of failures at the level of individual lines in a page. The OS reports failed lines when the runtime requests memory, and it may also communicate failures that occur during program execution.

The failure-aware memory manager never allocates live objects on failed lines and evacuates live objects from lines when they fail. We show how to minimize wasted space and performance degradation for garbage-collected managed languages, such as C#, Java, JavaScript, and PHP, using the hierarchical, *logical* line and block heap organization in the Immix mark-region garbage collector [3].

[PLDI '13, Seattle, WA, USA, 16 – 22 June, 2013.]

This heap organization reflects hardware cache lines and pages and simplifies failure tolerance because it manages memory in sizes similar to the failure granularity.

This basic software design delivers a correct, simple failure-aware approach for heap data and dynamically generated code with modest performance degradation for low failure rates. Even though hardware-unassisted software results in correct behavior, we show that memory fragmentation causes significant overheads when the number of failures rises above 10%. Fragmentation makes it too time consuming to locate free memory for medium and large object allocations. Novel, low complexity *failure clustering* hardware comes to the rescue by redirecting failures within a region (one or more pages) to a cluster at one end, greatly reducing virtual memory fragmentation. This modest hardware support solves the fragmentation problem for software, increasing the number of failures that are performance-transparent.

We implement this approach in a Java virtual machine, but it generalizes to any managed language. Since PCM is not yet available as a DRAM replacement, we add fault injection between the OS and VM memory allocators and execute on current processors with DRAM. Experiments with DaCapo benchmarks [5] show that failure-aware software adds no overhead in the absence of failures, and with 10% and 50% failure ranges on average adds only 4% and 12%, respectively. We explore the sensitivity of our approach to different clustering granularities and software line granularities. These results show that hardware failure clustering brings additional benefits because it creates larger contiguous pieces of working memory, even entire working pages, with little performance degradation. These results contradict accepted hardware wisdom on wear leveling [17, 18, 26]. Wear leveling delays any one failure, but once memory begins to fail, uniformly distributed failures cause fragmentation, whereas concentrated failures fragment memory less and are more transparent to software.

The capability of a managed runtime to relocate objects increases the ability of the system to tolerate uncorrected hardware failures, independent of the underlying hardware error correction mechanism. By allocating around holes in non-homogeneous memory pages, our system increases the useful lifetime of PCM memory, making it more practical to use as main memory.

2. PCM Background and System Design

This section explains why DRAM technology is struggling to scale to smaller technologies. It then briefly describes PCM technologies, approaches to extending PCM lifetimes, and how future systems may incorporate PCM. We use the following failure terminology. (1) *Static* failures occur prior to an application executing. The runtime may never allocate into pre-existing failed memory. (2) *Dynamic* failures occur during an application's execution.

2.1 Scaling Problems for DRAM

DRAM technology manufacturers are running into difficulties scaling DRAM cells to smaller features (thus denser memory chips) because DRAM uses a capacitive storage principle. Each cell is a capacitor, which stores electrons. The charge held by the cell is later sensed to determine the state stored in the cell. As cells get smaller, the charge held in each cell drops, which increases the probability that charge escaping from the cell will cause a bit flip. Emerging technologies such as PCM are much more stable than DRAM, so they became interesting potential replacement technologies.

2.2 PCM Hardware

The basic storage principle of PCM is changing the state of a chalcogenide material [19]. Electrical current heats up cells and then the material cools down into an amorphous or a crystalline state that have different resistances, which encode logical zeroes and ones. Due to the physical nature of state changes on writes, cells tend to wear out. The average lifetime is currently about 10^8 writes per cell versus 10^{15} in DRAM. These changes are isolated and localized to individual cells, so failures have no spatial correlation. The peripheral circuitry responsible for performing the write detects failures by reading the value once the write completes.

Previous work advocates for hardware and software wear leveling [17, 18, 26] that uniformly distributes failures across memory with the purpose of wearing it equally and delaying failures. Hardware-only error correction replaces bits or entire lines on permanent failures [16, 22, 23, 25]. These papers assume that once the finite error correction resources are exhausted, the entire page, if not the entire memory, fails. This was inspired by how DRAM systems handle failures today, but it is not adequate for PCM because failures are not as rare as in DRAM. This paper shows that, with modest hardware and software support, only the failing line needs to be disabled and removed from the visible memory space. Furthermore, software tolerates failures better when they are clustered, rather than uniformly distributed throughout memory.

Our software/hardware cooperative approach extends system endurance to go beyond what is possible with hardware-only error correction because it successfully uses other lines in the same page. Effectively, the system gradually shifts from error correction to error tolerance. An additional benefit is that error correction resources previously used to correct the failed line can be repurposed to correct lines that are still in use when they experience additional bit failures, extending the system lifetime even further.

2.3 System Design

Even though the bulk of a system's main memory may eventually be composed of PCM, we assume in this paper that the system will have some DRAM protected by regular ECC for operations that must not fail. For example, DRAM should store essential OS and runtime data structures. The lower the software's dependency on DRAM and perfect PCM, the more failure-tolerant it will be. One may wonder what incentive application writers have to use PCM when DRAM is also available. The right balance between PCM and DRAM sizes will depend on costs and goals of the system, but if PCM reaches the scale required to service the main memory market, it will have to be significantly cheaper than DRAM. As such, main memories are likely to use a small DRAM module and a much larger PCM module, making DRAM scarce. A process storing data in DRAM would have a higher probability of having its pages swapped out into disk, which will likely degrade performance. If the process stores data in PCM, performance improves because this module is large and its contents are less likely to be swapped out, if at all.

3. Cooperatively Avoiding Faults

This section describes our proposed hardware, OS, and runtime support for handling PCM failures.

3.1 Hardware Support

We propose two simple hardware mechanisms. The first informs the software of failures, and is the foundation for cooperative failure tolerance. The other clusters failures to reduce fragmentation.

3.1.1 Hardware: Failure Tolerance

The hardware cooperates with software by (a) informing software when a write fails, and (b) transparently maintaining the data from a failed write while the software adapts to the failure.

When a PCM write fails, the PCM memory module: (1) copies the data and the corresponding physical address to a failure buffer,

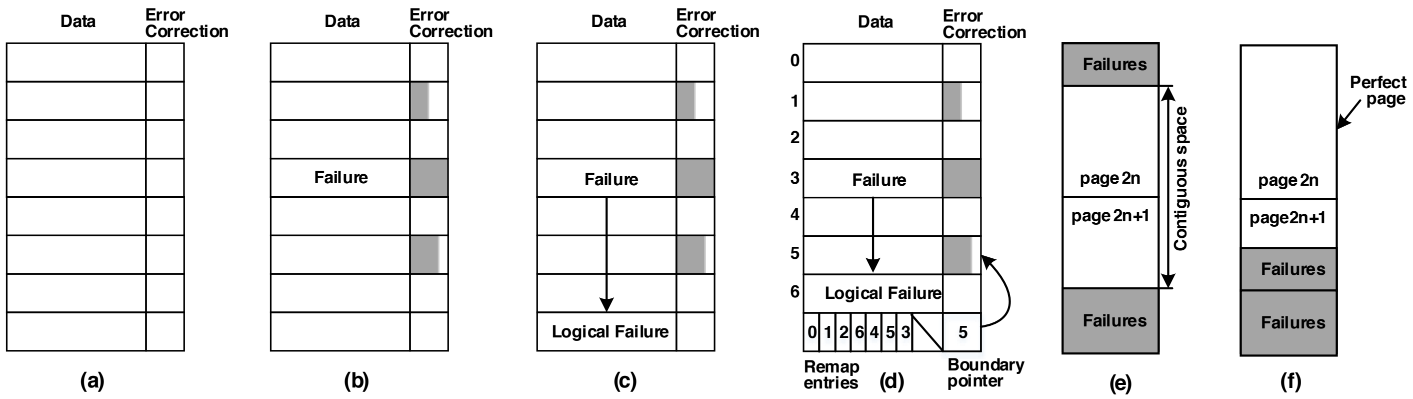


Figure 1. Hardware Failure Clustering

and (2) interrupts the processor. The failure buffer consists of a small quantity of SRAM or DRAM that is part of the PCM module itself, its peripheral DIMM circuit, or the memory controller. Every memory read request checks the buffer for the latest value written to a memory location. This search is performed in parallel with the actual access, so it does not affect read latency. If the memory module finds an entry in the buffer, it returns the corresponding data instead of the data read from the PCM array. The OS invalidates entries in this buffer once it finishes handling them. If other failures happen before a first failure is processed, information about these failures is stored in the failure buffer in FIFO order. The size of the buffer bounds the number of simultaneous failures that can be tolerated. The physics of PCM results in failures that are not naturally clustered in practice and do not happen frequently, so this buffer can be small (no larger than the processors' load/store queues, which have similar forwarding capabilities). An earlier entry with the same address is invalidated. When the failure buffer is about to fill up (enough entries are reserved to drain all outstanding writes to PCM), another type of interrupt is sent to the processor, and the PCM memory stops accepting any further write requests until the OS handles and clears at least one buffer entry. The hardware and OS thus prevent deadlocks and data loss.

3.1.2 Hardware: Minimizing Fragmentation

As failures accumulate, they increase memory fragmentation, making it more difficult to allocate data across multiple contiguous lines. We propose a hardware mechanism called *failure clustering* to mitigate this problem. Failure clustering logically pushes failures to one end of a region, consisting of one or more pages. Section 6.4 quantifies fragmentation due to failures and shows that reducing fragmentation greatly reduces software overhead.

Failure clustering logically remaps failed lines to edges of regions, as illustrated in Figure 1. Figure 1(a) shows a region with no failures. Figure 1(b) shows that, as the region wears out, lines run out of error correction resources and fail. Figure 1(c) shows that ideally these lines map to the edges of regions, maximizing the amount of contiguous space available for object allocation. To create contiguous space, we propose a simple line redirecting strategy. Figure 1(d) shows that every time a line fails, the hardware redirects it to the top or bottom of the region. Once the first line fails, the hardware installs a redirection map in the first line (or another pre-defined line) of the region and a pointer to the boundary between lines that work and lines that do not.

The redirection map has as many entries as lines in a region (128 by default in our experiments) and each entry requires as many bits as the logarithm of this number. Each entry is indexed by the physical address offset in the region and produces the actual line offset to which that line was redirected. On each failure, the hardware maps the failed address to the boundary pointer line, advances the boundary, and updates the redirection map, exchanging the offsets of the two lines being swapped. It increments or decrements the boundary, depending on which side of the region the failures cluster. If regions are page-sized, to further maximize the amount of contiguous memory, failures cluster to the top of even regions and to the bottom of odd regions, as shown in Figure 1 (e). Figure 1(f) illustrates regions larger than one page, in which case failures of both pages cluster in only one of them (as long as the number of unavailable lines corresponds to less than one page). Multiple-page clustering creates logically perfect PCM pages. Assuming a 4 KB page, 64 B lines, and a 2-page region size, the redirection map requires 889 bits, or two lines, for its metadata: 126 7-bit fields for redirection entries, and one 7-bit field for the boundary pointer. It uses the remaining bits for error correction.

When the first failure happens in a page, the hardware sets up a redirection map. The memory module first places a 'fake' failure at the location in which it intends to install the redirection map in its failure buffer, before inserting the entry corresponding to the first failure. Once the OS clears both entries from the list, the memory module installs the redirection map. The hardware handles failures in the redirection map with error correcting codes.

On memory operations, the memory module must identify which lines to redirect. The hardware uses a single bit in the original error correction metadata to indicate if the line is redirected. The memory module inserts the failure maps at fixed locations to ease lookup — the top of even regions and bottom of odd regions.

The implementation complexity of clustering hardware is modest. Upon an application reference to a region with failures, the memory module accesses the redirection map using the physical address offset it received from the cache hierarchy as the index. It retrieves a new offset and then accesses the corresponding line. In practice, this functionality requires very simple logic. However, it requires three memory accesses: (1) The first access finds the line is redirected, and (2) the second access goes to the redirection table, and (3) the third access reads or writes the redirected line. Three accesses could significantly increase latency. A simple solution is to cache recently used redirection maps. Note that in the common case of no failure, references require only one memory access irrespective of whether or not caching is in use.

3.2 Operating System Support

When the system is new, most, if not all, pages will be perfect (no failures). As PCM pages wear, their cells start to fail, and, at some point, hardware error correction is exhausted. Certain lines in a page become unusable, although other lines are still functional.

When a line fails, the operating system receives the interrupt notification of the failure. The memory access may have executed in a user process or the operating system itself. This paper focuses on the first case. The OS accesses PCM, as discussed below, and must handle situations in which its writes fail, but this paper does not consider storing OS data structures or code in PCM.

Operating systems can handle the failure in two fundamentally different ways. They can hide line failures from executing processes by replacing failed pages with perfect pages, changing the process's page tables and continuing the process without interruption. This approach masks the failure, preserving the illusion of perfect memory, at the cost of depleting the increasingly scarce resource of perfect pages.

The other approach is to notify the process of a line failure, to allow the runtime to reorganize data structures and avoid using memory locations in the failed line. The process continues using the page until its defect density renders it unusable. The focus of this paper is the design of failure-aware managed runtimes that may relocate data transparently to applications.

3.2.1 OS: Data Structures and System Calls

The OS manages DRAM, perfect PCM, and imperfect PCM pages in separate pools. Initially, all PCM pages start in the perfect pool. When a page first becomes imperfect, the OS moves the page to the imperfect page pool. The OS also tracks the failed lines in these pages. A system with 64 B lines and 4 KB pages requires a 64-bit bitmap per page. The OS stores these bitmaps in a table in DRAM with an entry corresponding to each physical PCM page in the system. Uncompressed, this table is approximately 1.6% of the size of the PCM page pool. Run-length encoding or other simple encoding techniques may provide high compression rates and reduce this overhead, especially when the system is new and the number of failures is low. When the system is shut down, the OS may save the failed line map to persistent storage and restore it on system initialization. Alternatively, the OS may rebuild the table by eagerly scanning memory or by lazily rediscovering failures at first write. Rediscovery is necessary after abnormal shutdowns and incurs a cost proportional to the size of PCM or the number of failures not recorded in persistent storage before the shutdown.

Failure-unaware processes may continue allocating perfect memory via normal mechanisms such as *mmap*. A failure-aware process can utilize both perfect and imperfect memory pages. Perfect memory grows scarcer as a system ages. Although imperfect pages require extra management and overhead, they are more abundant. A failure-aware runtime uses a special variation of *mmap* to acquire imperfect pages. This call returns the number of pages requested, however not all of the allocated memory may be usable. The runtime uses a *map-failures* call to get a failure map of the allocated memory region. If the runtime requires more space than the OS returned, it requests additional memory.

3.2.2 OS: Handling Dynamic Failures

When the memory module detects a failure, it raises an interrupt. This failure occurs asynchronously with respect to the write operation because a line is only written to PCM when it is evicted from the cache subsystem. When the interrupt is raised, the OS interrupt handler reads the failure information from the failure buffer. Before removing any entries from the failure buffer, the OS must prevent accesses to the failing addresses because, once it removes them, the memory module no longer forwards data to read operations on these addresses. The OS finds the corresponding virtual pages via reverse address translation, removes read and write permissions from them, and updates its own failure map. Reverse address translation is relatively expensive, but dynamic failures are very rare and either require copying a page or a garbage collection. The cost of reverse address translation is small compared to both.

The OS handler needs to resolve every failure. For failure-unaware processes, the only option is to copy the entire affected page to a perfect page. It must do the same for memory regions allocated when failure-aware processes request perfect memory. For imperfect memory requests, it may use the same approach or rely on the runtime to handle the failure.

A failure-aware runtime must register a handler with the OS before it uses imperfect memory. When a failure occurs, the OS performs an up-call on the handler, passing the addresses and data of all pending failures. The runtime must relocate affected data, updating whatever runtime structures are necessary to ensure correctness and transparency before returning from the handler.

As a convenience to the runtime, the OS may optionally reconstruct the affected page using a DRAM page for the duration of the runtime handler performing its work. Otherwise, the runtime uses the partially-failed PCM page and the information passed to the handler to reconstruct the affected data, and must avoid accessing the failed lines while it handles the failure.

3.2.3 OS: Paging and Other Issues

The OS may occasionally copy data from one imperfect page to another, for example, when an imperfect page is swapped out and then brought back into memory. If the failures on the destination page are a subset of the failure locations on the source page, copying is straightforward. However, keeping track of page compatibility is expensive, and prior work shows similar matching processes have limited efficacy in practice [11]. As we show below, failure clustering helps solve this problem.

When moving data from a previous imperfect page (possibly on disk) to another memory page, the OS has three options: (1) The OS may swap data into a perfect page. (2) The OS may swap data into an imperfect page with different failures compared to the previous physical page. The OS must inform the runtime of the new failure map by delivering an up-call on the runtime's handler. If the runtime cannot safely move data in the failed lines, the OS can try another imperfect page or fall back to a perfect page. (3) Failure clustering enables a third and simpler approach. Since clustering accumulates failures at one end of the page, the OS simply maps the page it is swapping in onto any available page with the same number or fewer failures.

3.3 Runtime and Application Software Support

We turn now to failure-aware runtimes. We start with the limitations inherent to native (unmanaged) applications. We then overview failure-aware runtime support for managed applications. Section 4 describes the basic Immix garbage collection algorithm and how we implement failure-tolerance in Immix.

3.3.1 Failure-Aware Applications and Runtimes

In principle, any application can be made failure-aware. In practice, forcing developers to manage failures without runtime support is impractical because it requires significant and pervasive changes to ensure live data never occupies failed memory.

Static failures to memory allocated to the *stack* for both native and managed languages could be addressed by the compiler and runtime stepping around failed lines as they allocate new stack frames. But given the frame allocation/deallocation frequency, the overhead of such a scheme is likely to be unreasonable. Handling dynamic failures to the stack is even more challenging.

Static failures to memory allocated to the *code and data segments* will be hard to handle because of hard-coded assumptions about offsets to code and data in these segments. In principle, this problem could be addressed via binary rewriting. However, because native programs may hold untyped and undiscoverable references to elements in the code and data segments, handling dynamic failures to these regions appears to be intractable.

The runtime memory manager may handle static failures in *heap* memory. Native runtimes use free list allocators, which could use imperfect memory, and mark as unavailable those units of allocation that coincide with failed memory. Such an implementation would increase the complexity of managing the free list. However, for large objects and large numbers of failures, native runtimes have a stronger requirement for perfect pages than managed runtimes. For example, managed languages can split large arrays [21] (see Section 4), whereas native languages require contiguous allocation.

Adding failure tolerance to the free-list used by native runtimes will incur additional complexity, fragmentation, or both. Most of the complexity arises due to (1) mismatches between failure granularity and the free-list heap layout, and (2) the need to differentiate between failed and in-use memory both to reduce fragmentation and for correctness with conservative collection. Granularity mismatches arise because native memory managers use segregated size free-list allocators [2, 6, 8, 14], carving up large fixed-sized regions (e.g., one or more pages) into same-sized objects on demand. To add failure tolerance, we have three choices: (a) to use only perfect small page size regions, which incurs fragmentation for medium size objects, and never uses pages with failures, or (b) incur fragmentation in each fixed-size large region, or (c) add complexity to allocate and recycle variable-sized regions. Note that these same issues arise for managed languages that use free-lists. None of these options are appealing.

To summarize, native applications could be made tolerant to static failures in the heap with non-trivial changes to a free-list allocator, but the OS must still handle all dynamic failures.

3.3.2 Failure-Aware Managed Runtimes

We design and implement a solution for failure-aware memory in a *managed* runtime that dynamically allocates code and data memory space. A large and growing set of applications are written in managed languages, such as C#, Java, JavaScript, Python, and PHP. These languages use safe pointer disciplines and execute in virtual machines (VMs) that include dynamic compilers and automatic memory management (garbage collection). Because these languages are memory safe, the garbage collector (GC) allocates and is free to move objects in memory (including code objects in some VMs). Moving objects is both safe and transparent to applications. We leverage this functionality to tolerate failures in PCM without putting a burden on developers. A failure-aware garbage collector never allocates objects in failed memory and relocates affected data when new failures happen.

In VMs that dynamically allocate code and data, our solution transparently handles both, but we focus on data here. A VM solution reduces the need for perfect memory to the VM itself, heap metadata, and large objects. A self-hosting VM may reduce the need even further. The next section provides an overview of how to make garbage collection failure-aware.

3.3.3 Garbage Collection

Two high-level modifications are necessary for a garbage collector to tolerate failures: (1) exposing information about failures to the garbage collector (GC), and (2) modifying the allocator to avoid allocating live objects in memory that contains failures.

The OS communicates a failure map to the GC. The GC adds the failure map to the heap metadata that records which memory contains live objects and which memory is free for allocation. The hardware determines the finest possible granularity of the failure map. This paper uses the cache line size as the finest failure granularity, which is also typically the same granularity of hardware write operations to PCM. The failure map may use coarser granularities that trade-off less storage overhead for less available memory as failures accrue. We evaluate this trade-off in Section 6.3. For simplicity, we assume here that the failure map and other heap metadata are stored in perfect memory and do not themselves fail.

In a failure-aware VM, (a) the GC must never allocate live objects in a failed region of memory, and (b) when a region of memory fails during execution, the collector must evacuate all objects in the failed region and allocate them elsewhere.

Static failures The GC already maintains the invariant that it only allocates objects into free memory. The GC treats all allocations the same regardless of whether an object allocation is the result the application's call to `new()`, or if the GC itself is allocating an object to evacuate (move) it during a collection. A failure-aware GC maintains the same invariant. It only allocates into free memory. It treats failed memory the same as memory with live objects on it (neither are free). With a static failure map in hand, the collector simply allocates into free memory.

At a minimum, all memory managers separately manage (i) small or medium objects and (ii) large objects. Given sufficient failures and a large enough object, imperfect memory will not accommodate a large object. As with any allocation, when the failure-aware memory manager first attempts to allocate an object and there is insufficient memory, it triggers a collection to reclaim free memory and then attempts to allocate again. If the GC still cannot accommodate the object in imperfect memory, the allocator requests perfect memory. We greatly reduce the probability of this situation with hardware clustering at the granularity of two or more pages to create logically perfect pages (see "Large Objects" below).

Dynamic failures If a failure occurs during program execution, the OS may notify the GC and provide it with the values the program intended to write into the affected region. The OS can specify dynamic failures at the granularity of lines or pages. Regardless, the collector must move any affected objects, allocating them elsewhere in the heap. This requires that all pointers to the affected object(s) be identified and retargeted to the moved object(s). In a typical generational collector, a failure to memory in the nursery will only require a nursery collection, but any other failure will require a full heap collection. If no live objects reside in the affected region, which is unlikely because writes cause the failures, the collector marks the memory as failed in the failure map and returns.

Pinning support is required in some languages, such as C#. If a live object in the failed region is pinned by the application, the VM cannot move it and must notify the OS that it cannot vacate the region. An alternative solution is to allocate pinned objects to memory that does not fail, but some systems require in-place pinning after allocation, so this solution will not always work. Since pinning is rare, the probability of a line failing that contains a pinned object is low and some expensive action, such as the OS remapping the affected page to another perfect or failure-compatible physical page would be appropriate.

While the collector recovers from a failure, other failures could happen. As explained earlier, the hardware and OS handle these failures until the collector is ready to deal with them.

Heap Layout To implement failure tolerance, the garbage collector must relate the PCM failure granularity to the data structures it uses to manage live objects and free memory. Modern garbage collectors use one of contiguous allocation, a free-list, or the Immix mark-region line and block hierarchy [3, 4]. The next section describes how to use the Immix heap, and Section 3.3.2 described why modifying a free list is possible, but complex. Modifying a contiguous heap layout is not even possible, because it simply has no way to skip over or manage small regions of memory without fundamental changes to the algorithm.

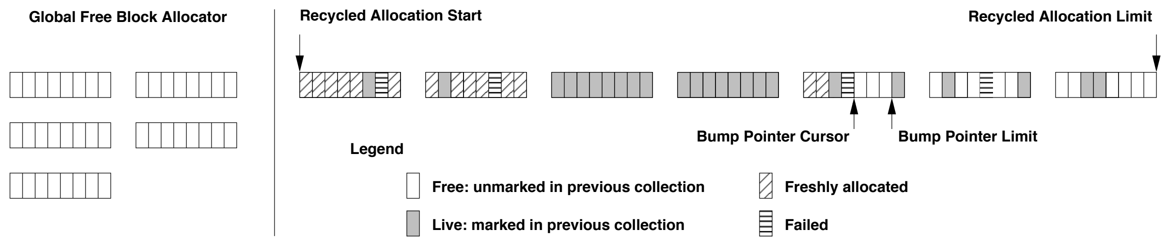

Global Free Block Allocator
Recycled Allocation Start
Recycled Allocation Limit
Bump Pointer Cursor
Bump Pointer Limit
Legend
Free: unmarked in previous collection
Freshly allocated
Live: marked in previous collection
Failed

Figure 2. Immix Heap Organization and Amenability to Failure Tolerance

Large Objects Both memory managers and garbage collectors explicitly handle large objects separately [2, 4, 14, 21]. The exact large object size is a function of the granularity of the memory manager's block size, i.e., the granularity at which it manages the small and medium size objects. Typical block sizes range from 16 to 64 KB. Immix uses 32 KB by default. Most large objects in managed languages are arrays and, for DaCapo Java programs, they consume on average half the heap [21].

Without hardware clustering, if many failures are uniformly distributed in memory, large objects will not fit. A purely software solution is to use discontiguous arrays, which divide up arrays into a spine that points to smaller fixed-size *arraylets* and modify the array access code [1, 21]. Discontinuous arrays were invented to bound pause times for real-time systems. Sartor et al. showed how to make them relatively efficient, with average overheads below 13%, even with arraylets as small as 256 bytes.

With hardware clustering at a two-page granularity or higher, the hardware creates the logically perfect pages that large objects require. For example, with two-page clustering and failures in roughly 50% or less of the memory, at least 50% of memory will consist of perfect pages that the allocator may use for large objects.

4. Implementing a Failure-Aware Runtime

This section describes the modest changes required to make the Immix mark-region garbage collector failure-aware. The key attributes of Immix are well suited to the PCM failure mode, in which fixed-size lines fail. (a) Immix already tracks memory at a line granularity categorizing lines as: free, live, or live pinned. (b) Immix already maintains two invariants: (*i*) it only allocates into free lines, and (*ii*) it only ever moves unpinned objects. To make Immix failure-aware, we simply add a fourth line category to Immix, failed lines, and continue to enforce the same two invariants.

4.1 Basic Immix

The Immix design was motivated by the tension between three desirable performance objectives: space efficiency, collection time, and mutator (program) locality [3, 4]. Immix achieves all three performance objectives by combining bump-pointer allocation, efficient tracing, and occasional copying of live objects, as well as a memory-efficient two-level line and block heap organization. A key to this design is a bump-pointer allocator that can very efficiently skip over unavailable lines. This organization makes Immix an excellent match for our requirement that the allocator skip over failed PCM lines. Prior work established that Immix outperforms prior algorithms and heap organizations, and it is now the default collector in Jikes RVM [3].

Lines and Blocks Immix manages heap memory at coarse and fine granularity by dividing memory into blocks and blocks into lines, as depicted in Figure 2. Lines approximate cache lines and blocks approximate pages. Objects may span lines within a block, but cannot span blocks. Objects that fit within one line are regarded as *small*. Immix has a threshold that designates objects as *large* and delegates them to a separate page-grained large object space (LOS). This threshold is never larger than a block. Initially all blocks are free and Immix allocates objects contiguously into blocks until it fills the heap, which triggers a collection.

Collection Immix collection marks live objects by performing a transitive closure over the live object graph starting at the roots (global and stack variables) and then tracing references. When it encounters an object for the first time, it marks the object and its line as live. Immix does not copy objects by default; it deals with fragmentation by copying objects on occasion. Some managed languages, such as C# and JavaScript, require object pinning, in which applications can specify that an object may not be moved. This is typically supported for performant interoperability with native C. Immix respects pinned objects and never copies them.

After marking, Immix scans line mark tables, recycles partially filled blocks, and returns completely empty blocks to a global pool of pages for use by the whole runtime.

Allocation In steady state, Immix allocates objects into available recycled blocks first, as depicted in Figure 2. The allocator is parallel, with one thread-local allocator for each application thread. The allocator initializes a bump pointer to the first free line in a block and sets the limit to the end of the last free line in this contiguous set of free lines. When the bump pointer finds that an allocation request cannot be satisfied within the limit, it skips over any unavailable lines to identify the next set of free lines. When it exhausts a recycled block, it requests another from a shared pool of recycled blocks. Once all recycled blocks have been exhausted, the allocator requests completely free blocks from the global block pool. Once the global pool is exhausted the allocator triggers a collection. Other allocators, such as the large object space (LOS) compete for blocks from the global pool.

To make efficient use of recycled blocks, and avoid skipping otherwise usable smaller spaces, Immix heuristically allocates medium objects (those larger than a line) on a special overflow block if the object cannot immediately fit in the contiguous space available to the bump pointer. Overflow blocks are sourced from the global pool of completely free blocks, so have maximal contiguous space. Our failure-aware collector handles requests for completely free blocks by the LOS and overflow allocator by using perfect memory for such requests (see Section 3.3.3).

Sticky Immix The Sticky Immix algorithm adds high-performance generational behavior to the Immix collector. It combines sticky mark bits collection [8] with the Immix algorithm. Sticky mark bits collectors identify young objects by a bit in their header, rather than by allocating them into a distinct space. By default, it *opportunistically* copies nursery survivors into any available empty or partially filled blocks. If there is no available memory, it leaves them in place. This collector retains all of the attributes of Immix which make it amenable to failure tolerance and to the need to skip over failures, whilst gaining the performance of compacting and collecting younger objects first.

Performance Figure 3 shows a sample result that motivates Immix as a performant baseline for a failure-aware runtime. It shows the geometric mean of total execution time of the DaCapo benchmarks configured with full-heap mark-sweep (MS), Immix (IX), and the Sticky generational variants (S-MS and S-IX). We extend Immix to handle failures because of its high performance and its line-block organization simplifies the algorithmic design.

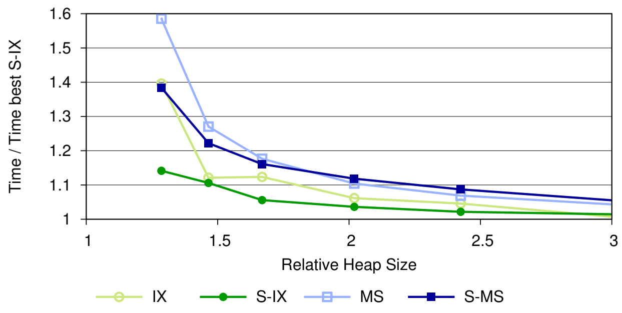


Figure 3. Performance of DaCapo benchmarks running with various memory management algorithms at multiple heap sizes.

4.2 Failure-Aware Immix

This section describes how we modify Sticky Immix to be failure-aware.

Static Failures When the OS gives Immix a region of memory and a failure bit map, failure-aware Immix marks corresponding lines as failed in its line-mark metadata. Because line marks are bytes and only some of the 256 available states are used, we add an additional state that denotes a failed line without space overhead. One subtlety is that the Immix line size is not necessarily the same as the PCM line size. For example, the best performing line size for Immix is 256 B without failures, whereas PCM line sizes are typically 64 B. When the Immix line size is greater than the PCM line size, we must mark the entire Immix line failed, even though only part of it failed. Section 6.3 explores this trade-off.

The previous section described how Immix uses free blocks for overflow allocation to accommodate medium sized objects that do not immediately fit in the space available at the bump pointer. With failure-aware Immix, the overflow block is not guaranteed to be perfect and the object may not fit. Thus we change the algorithm to search the remainder of the overflow block for suitably sized available space. If this search fails, the allocator resorts to requesting a free perfect block. Hardware failure clustering greatly reduces the likelihood of multiple overflow allocation failures.

Dynamic Failures To address dynamic failures, we use the same mechanism that Immix uses to defragment the heap. Immix opportunistically copies objects that reside on sparsely populated pages to free or partially free blocks during a defragmenting collection. For dynamic failures, we straightforwardly reuse this mechanism to move objects affected by dynamic PCM failures. We mark the affected object(s), the lines that contain it, and its block for evacuation, and then invoke a copying collection. Although a full heap collection is relatively expensive, dynamic failures are rare.

The full heap collection time is an estimate of the time needed by the runtime to handle a failure. For the DaCapo benchmarks on Intel Core i7 hardware, it takes 7 msec on average. The worst case is 44 msec (hsqldb). The next two are 22 and 12 msec (fop and xalan), with all others under 10 msec. To put this time in context, the average number of garbage collections is 14.7 and the average total execution time for these benchmarks is 1817 msec. We believe that for most applications, this simple strategy is sufficient for dealing with the rare occurrence of a dynamic failure.

5. Experimental Methodology

This section justifies our use of execution time results and then describes the hardware, benchmarks, collector configurations, and methodology for running experiments.

At the time of writing, direct evaluation on systems with PCM main memory is unfortunately not possible, as the only commercially available PCM parts today are NOR flash replacements, with very low capacities and incompatible physical interfaces. Simulation is slow, making whole benchmark execution prohibitively long. Instead, we evaluate our approach running natively on DRAM, and instrument the managed runtime with a fault injection module between the OS memory allocator and the VM memory allocation module. When the latter allocates memory, part of this memory is made 'defective' by the fault injection module.

Machine, benchmarks, and collector configurations All experiments execute on an Intel Core i7 2600 machine with 4 GB of DRAM main memory, running on Ubuntu 10.04.1 LTS. We use an implementation of Immix in Jikes RVM, a Java virtual machine. We execute the superset of all benchmarks in the DaCapo 9.12-bach and DaCapo-2006-10 suites that can run on Jikes RVM. We use lusearch-fix, which is the lusearch benchmark patched to fix a bug in the underlying lucene library that introduces pathological allocation behavior [24] by needlessly allocating a large data structure in a hot loop. This bug results in an allocation rate a factor of three higher than any other benchmark. Aside from reporting it in Figure 4 for completeness, we exclude the buggy version of lusearch from all of our analysis.

We compare to the Sticky Immix (S-IX) implementation in MMTk from Jikes RVM 3.1.2 described above with a block size of 32 KB and a line size of 256 B. Failures and garbage collection expose application performance to a space-time trade-off that we explore explicitly across a range of heap sizes. We use a modest default heap size that is 2× the minimum for each benchmark.

Failure map generation and memory accounting We model PCM failures via a failure map. The failure map has one bit for each 64 B PCM line, which indicates whether that line is working or has failed. The failure map generator distributes failures uniformly from a command line argument and produces clustered failure map distributions from uniform distributions.

Because the time-space trade-off is central to garbage collected systems, we explicitly control for heap size in all of our experiments. This introduces two important and distinct methodological considerations in the context of memory that may fail. First, we compensate the fixed heap sizes according to the failure rate to ensure that the *usable* memory is held constant. Section 6.2 discusses heap compensation in detail. Second, and quite separately, application memory requests now fall into two categories: those from *relaxed* allocators that are robust to failure because they can utilize fragmented pages, and those from *fussy* allocators that require perfect PCM or DRAM pages (because their allocations are page-grained). The balance between fussy and relaxed allocation varies greatly as a function of time and among benchmarks.

Because our modeling of PCM failure is not guaranteed to return a perfect page and a real implementation would have a limited supply of DRAM for such requests, we use a debit-credit based cost model to account for the cost of using a DRAM page when no perfect PCM page is available. Penalizing the use of DRAM is appropriate both because it would be scarce in practice and because, without a penalty, DRAM is highly attractive since it is never fragmented, which may lead to the counter-intuitive situation where higher failure rates offer better performance.

When a fussy allocator requests a perfect page and the allocator has sufficient memory, but no perfect page is available, we model giving it a perfect page (DRAM or PCM) with a one page space

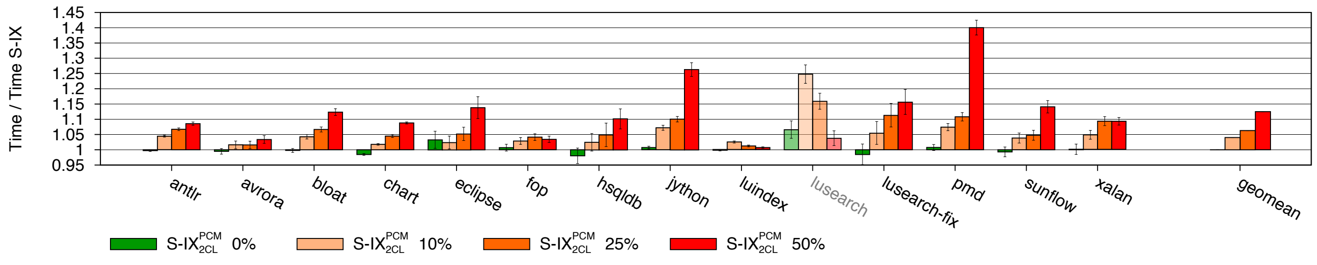


Figure 4. Performance of DaCapo benchmarks running with a failure-aware Immix implementation at multiple failure levels, normalized to the unmodified version of the algorithm.

penalty. Because garbage collection exists as a time-space tradeoff, this space penalty ultimately translates to a time penalty. The relaxed allocator may 'repay' the debt of the fussy allocator. On subsequent allocations, which may follow a collection or not, each time the relaxed allocator is given a perfect page, it only takes the page if there is no outstanding debt. If there is a debt, we reduce the debt by one page and fetch another PCM page for the relaxed allocator.

Nondeterminism and experimental error To eliminate the effects of nondeterminism due to timer-driven adaptive recompilation, we used the current version of Jikes RVM's replay compilation mechanism [10, 13]. Compiler replay is driven by optimization profiles gathered ahead of time for each benchmark and then used for all experiments. Each profile identifies a good optimization plan, stating the optimization level (if any) for each method, and includes edge counts and a call graph which would normally be gathered dynamically. The replay mechanism first runs each benchmark for one iteration without any optimization (this forces all classes to be loaded). The system then compiles all methods according to the optimization plan, before commencing a second iteration of the benchmark. Our experiments evaluate this second, optimized, iteration. This methodology eliminates the chaotic nature of timer-based adaptive optimization while delivering performance slightly better than steady-state performance for an adaptively optimized system.

We perform 20 invocations of each benchmark, each using the replay methodology, and take the mean of these results, reporting 95% confidence intervals. The 95% confidence interval error is generally very low, around 1-2%. We aggregate results across benchmarks using geometric means. Some configurations cannot execute some of the benchmarks in very small heaps. When reporting aggregated results, we discard results at a given heap size when some of the benchmarks do not complete, which manifests in the graphs as lines that terminate before reaching the y-axis.

6. Evaluation

This section starts with an evaluation of PCM-aware Sticky Immix implementation with failure clustering hardware. It shows how together they dramatically mitigate the effect of PCM failures. The remainder of the section explores trade-offs in the design space and illustrates the effects of failure-induced memory fragmentation.

6.1 Performance and Memory Overheads

Figure 4 shows the performance overheads of failure-aware Sticky Immix with two-page failure clustering (S-IX$_{2CL}^{PCM}$) at multiple PCM failure rates (0, 10%, 25% and 50%). The figure normalizes time to the unmodified Sticky Immix (S-IX) collector (not shown). All of the Immix collectors use a logical line size of 256 B and block size of 32 KB, and receive the same amount of *usable* memory (the equivalent of two times the minimum heap required by each benchmark to run). Note that S-IX$_{2CL}^{PCM}$ incurs zero overhead when there are no PCM failures (the geometric mean of the green bars is 1.0, the same as unmodified Sticky Immix). The lack of any measurable overhead corroborates that, as described in Section 4.2, no additional metadata is required for failure-aware Sticky Immix (or we would see slowdowns at this constrained heap size).

As the number of failures increases, the performance overheads slowly increase. At failure levels of 10%, the overhead is 3.9% on average and never exceeds 7.4% (peach bars). Low overheads are a result of the positive effects of failure clustering, which are more thoroughly explained in Section 6.4. The overhead grows modestly with failure rate, increasing when the failure rate is 50% to an average of 12.4% with a maximum of 40% for pmd.

Some workloads, such as pmd and jython, experience high overheads. The reason is that they allocate many medium sized objects, which makes it more challenging to find free memory to fit these objects. Benchmarks such as xalan, which predominantly allocate very large objects, are quite resilient to failures because they utilize the entirely free pages delivered by two-page failure clustering. Virtual address translation transparently removes any problem of page-level fragmentation. As anticipated in Section 5, the buggy lusearch benchmark (which we gray-out here and exclude from all further analysis) gives a counter-intuitive result, with overhead *reducing* as failure rate rises. This result is due both to lusearch's pathological behavior, and an unobvious interplay between the two-page failure clustering and the way we use heap space compensation in our analysis (see Section 6.2 and Figure 5).

The remainder of this section explores the fragmentation problems and the inherent trade-offs due to failures in more detail.

6.2 Memory Reduction vs. Fragmentation

Three main factors impact performance as memory degrades. (1) As failures accumulate, the functional memory a VM receives when it requests a certain region of memory from the OS is reduced. This problem is addressed by compensation: the VM simply requests more memory until it gets as much working memory as originally intended. (2) Failures cause heap fragmentation, which reduces the usability of the available memory. (3) A more subtle effect is *false* failures, which occur when the PCM failure granularity is less than the granularity of allocation (similar to false sharing in caches). For example, if the Immix line size is greater than the failed cache line size specified by the PCM memory or the OS. With an allocation granularity of 256 B lines in Immix, an entire line becomes unusable when a single 64 B PCM line fails and false failures overstate the real failure by 192 B. This phenomenon is not specific to Immix, and will manifest whenever a request is made for contiguous memory greater than the PCM line granularity (regardless of the software algorithm).

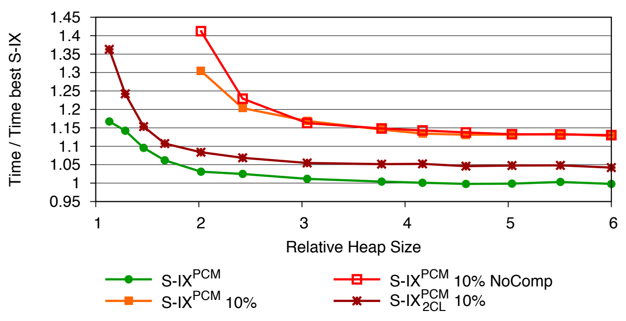


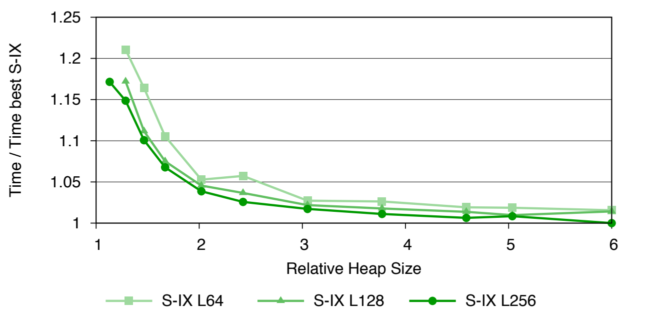
Figure 5. Compensation for memory failures is most noticeable at smaller heap sizes. This graph compares the baseline S-IX^{PCM} with no failures (green) to 10% PCM line failure rate and no clustering, with (red) and without (orange) memory compensation, and to S-IX^{PCM} with clustering and compensation (maroon), the best failure-tolerant configuration.

To break down these three effects, Figure 5 summarizes the behavior of different configurations by presenting normalized execution time geometric means (y-axis) over all workloads as a function of heap size (x-axis). It compares the baseline (green) with no failures (S-IX^{PCM}) to PCM-aware Sticky Immix with compensation (orange) and without (red), both *without* failure clustering hardware at a 10% failure rate (S-IX^{PCM} 10%). Space compensation provides the same amount of working memory if there were no failures. For example, given a heap size h used in the absence of failure, when the failure rate is f, we compensate with a heap size of $h/(1-f)$. Compensation ensures that the number of bytes of non-faulty memory available to the system is held constant.

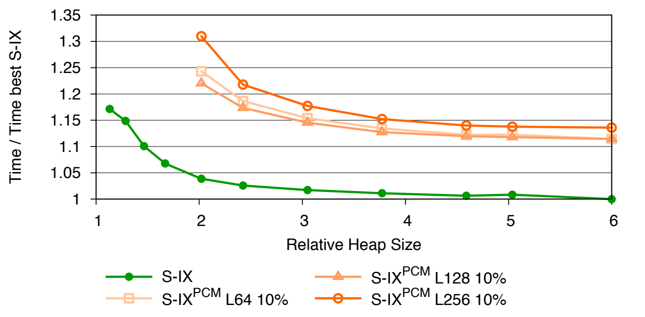
Consider S-IX^{PCM} under no failures (S-IX^{PCM}, green), at 10% failure rates (S-IX^{PCM} 10% NoComp, red), and with working-memory space-compensation at 10% failure levels (S-IX^{PCM} 10%, orange). The gap between S-IX^{PCM} 10% NoComp and S-IX^{PCM} 10% (red and orange) in Figure 5 shows the effect of failures reducing available memory. In small to moderate heap sizes, this effect is pronounced until the heap grows to 3 times minimum, when the lines converge. The gap between the S-IX^{PCM} 10% (orange) and no failures (green) on the bottom shows the remaining two effects. Fragmentation and false failures, when spread uniformly over memory, significantly degrade performance, over 25% in small to moderate heaps and over 10% even in large heaps.

The large effects of fragmentation with or without heap compensation motivate hardware clustering. Comparing no clustering (the top two lines) to S-IX$^{PCM}_{2CL}$ 10% (two-page clustering, maroon) shows that the latter significantly improves performance by reducing the amount of fragmentation exposed to the software. Reducing fragmentation wins twice. First, the amount of usable memory is increased, which reduces memory management load, shifting the curve left. Second, application locality is improved, shifting the curve down. The next section examines in more detail the effect of line size on false failures.

Heap compensation can have an unexpected interplay with two-page failure clustering and our means of accounting for requests for perfect memory, as in lusearch in Figure 4. When allocation is dominated by page-grain requests, the number of failures on a page is unimportant; what matters is whether the page is completely free or not. Consequently, the impact of failure rate on such benchmarks is essentially constant for non-zero failure rates. On the other hand, heap compensation makes more memory available as failure rates grow. Thus although in practice the cost in terms of usable pages for failure rates of 10%, 25% and 50% remains similar for a benchmark like lusearch, compensation increases, which leads to the counter-



(a) The effect of line size on baseline S-IX.



(b) The effect of line size on S-IX^{PCM} with 10% of lines failed.

Figure 6. The presence of failures without hardware clustering alters the performance behavior as Immix line size grows: While S-IX uniformly benefits from larger lines, in the presence of failures, S-IX^{PCM} L256 suffers.

intuitive improvement in performance as failure rates grow that we see in the lusearch result in Figure 4. However, this pathological behavior is rare, as Figure 4 shows.

In the remainder of the evaluation, we use compensated runs by default.

6.3 The Effect of Line Size

Figure 6 shows the effect of Immix line size on performance, assuming a constant PCM line size of 64 bytes and no hardware failure clustering. Figure 6(a) corresponds to the baseline S-IX at three Immix line sizes: 64 bytes (S-IX L64), 128 bytes (S-IX L128), and 256 bytes (S-IX L256), all running without PCM failure. The interesting trend to note is that larger lines perform better on S-IX, and that this advantage increases as the heap becomes smaller. The primary reason is that in a tight heap, larger lines mean fewer slow-path operations and better locality. Smaller lines increase metadata overhead, which is felt most acutely in constrained heaps.

The presence of failures changes the Immix line size dynamics significantly. Figure 6(b) shows the same three Immix line sizes (S-IX^{PCM} L64, S-IX^{PCM} L128, and S-IX^{PCM} L256) at 10% failure levels, as well as S-IX. False failures make larger Immix line sizes less desirable when we constrain the heap without clustering. As the Immix line size increases, it takes only one PCM line failure to make an entire 128-byte or 256-byte line unusable. These false failures generate more severe loss of usable memory space, which affects performance negatively. Once again, the issue is alleviated, but not eliminated, at larger heap sizes.

Figure 7 further illustrates the relationship between performance, Immix line size and failures, this time holding heap size constant at 2× the minimum and varying failures between 0 and 50% of all lines. To more fully expose fragmentation behavior, these configurations do not use hardware failure clustering. In this

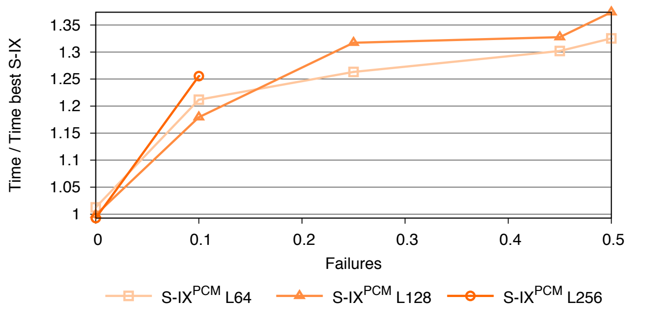


Figure 7. The effect of PCM failures on three different Immix line sizes at a fixed 2× heap size. With a 256 B Immix line, failures dramatically reduce useful memory causing benchmarks to fail without hardware clustering.

case, the larger the line size, the earlier it starts causing excessive false failures and thus wasting memory, so the worse the performance. When the failure rate is zero, larger lines perform better, but as the failure rate increases, the effect of false failures dominates. With L256 this happens almost immediately, and with L128 the crossover is at about 15% failure rate.

Figures 6 and 7 illustrate the two opposing forces that influence the choice of Immix line size without clustering: lower memory management overhead versus higher usable memory waste.

6.4 Failure Clustering

The results in Section 6.3 assume that PCM failures occur with a uniform random distribution at the granularity of 64 B lines. When uniformly distributed, these failures cause significant fragmentation, and will lead to as much as 3× space overhead due to false failures with a 256 B Immix line. This section quantitatively explores the effect of clustering failures at granularities greater than 64 B. The number of failed lines remains 10%, 25%, and 50%, but we systematically distribute failures in clusters of 2^N failed lines, for sizes 64 B and greater to reveal the effects of fragmentation and how well clustering counterbalances them.

Failure Clustering Limit Study To conduct these experiments, we use the S-IX^{PCM} collector, but generate the PCM failure map slightly differently. Instead of failing individual 64 B lines with probability p, we step through aligned regions of size 2^N and fail the entire region with probability p. The effect is that the gaps between failures are guaranteed to be at least 2^N, but the probability of any given line having failed remains p. This analysis motivates hardware support for failure clustering.

Figure 8 shows performance when S-IX^{PCM} is exposed to 10%, 25%, and 50% failures with failures clustered at powers of two from 64 B to 16 KB. The x-axis is logarithmic. Performance is normalized to unmodified S-IX running on regular memory.

Performance dramatically improves with failure clustering. Clustering mitigates fragmentation by reducing or eliminating false failures and leaving larger chunks of usable memory for the memory manager to use. The greater the number of failures, the more dramatic the effect of failure clustering. The problem is so severe that starting at 25% failures, and at 64-byte failure cluster granularity, the VM cannot execute many workloads to completion even at a 2× heap, thus the 25% and 50% curves start at 128 bytes. Yet, using clustering at 256 bytes, the performance overhead is reduced to just 20% when 50% of lines fail.

Proposed Failure Clustering Hardware Figure 9 compares our failure-aware collector with and without clustering hardware support at one- and two-page granularity, for 64 B, 128 B, and 256 B Immix lines, and with 0, 10%, 25%, and 50% of PCM lines failed.

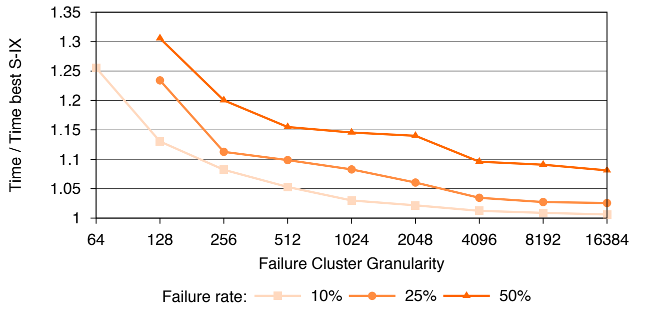
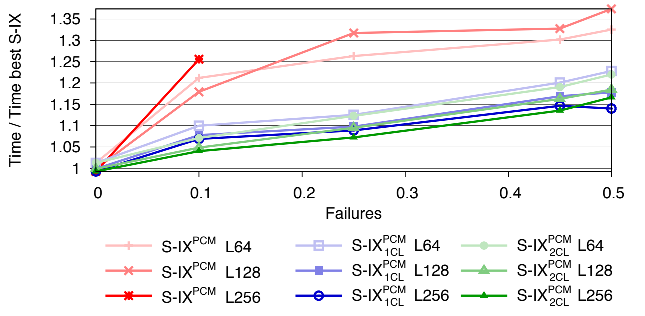
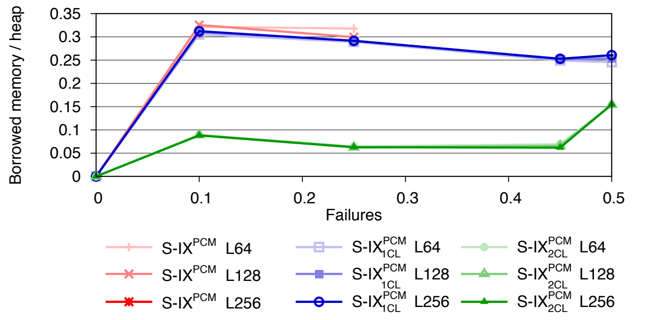


Figure 8. S-IX^{PCM} performance with default 256 B lines as failures are clustered at a variety of power-of-two granularities (x-axis) with 10%, 25% and 50% of lines failed. Clustering of failures dramatically mitigates performance penalties of failures.

These experiments use a failure map with uniformly distributed 64-byte line failures, and then move those failures according to our one- and two-page clustering algorithm, alternatively moving all failures to the start or end of each clustering region.



(a) Hardware failure clustering greatly improves performance.



(b) Two-page clustering greatly reduces dependency on borrowed pages.

Figure 9. Hardware support for failure clustering mitigates fragmentation, reduces or eliminates false failures, and greatly reduces demand for perfect pages.

Figure 9(a) shows the effect on average performance, while Figure 9(b) shows the impact on average demand for perfect pages. The first group of curves (red: S-IX^{PCM} L64, S-IX^{PCM} L128, and S-IX^{PCM} L256) does not have hardware support, while the second (blue: S-IX$^{PCM}_{1CL}$ L64, S-IX$^{PCM}_{1CL}$ L128, S-IX$^{PCM}_{1CL}$ L256) has one-page clustering and the third (green: S-IX$^{PCM}_{2CL}$ L64, S-IX$^{PCM}_{2CL}$ L128, S-IX$^{PCM}_{2CL}$ L256) has two-page clustering.

Figure 9(a) shows that hardware failure clustering greatly reduces the performance overhead due to PCM failures. The first group of lines (red, top) consistently perform worse because frag-

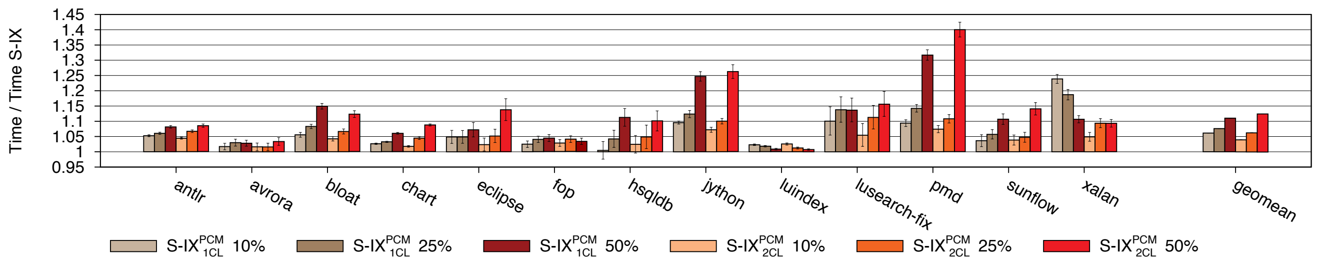


Figure 10. Performance of failure clustering at one- and two-page granularities for each of the DaCapo benchmarks.

mentation and false failures dominate performance, especially when lines are large and many lines fail. The 256 B line size fares worst, and the effect of false failures is so great that many benchmarks cannot run at 25% failure (hence no data for that point in the graph). Performance is much better and less variable with hardware support for failure clustering because clustering eliminates most false failures and greatly reduces fragmentation visible to software.

Two-page clustering (green) has the further advantage of creating completely failure-free pages. Note that for both one-page and two-page clustering, 256 B Immix lines result in the best performance. The reason is that clustering mitigates the negative effects of fragmentation, leaving only the positive effects of coarser Immix line granularity (lower memory management overheads).

Figure 10 shows the individual benchmark results for one- and two-page failure clustering at various failure rates. Consistent with Figure 9(a), two-page clustering reduces overheads considerably unless failure rates are very high. Figure 9(b) shows that the main reason is that hardware failure clustering greatly reduces demand for perfect pages. This reflects the fact that clustering has a defragmenting effect, which greatly increases the probability of a medium sized object being able to fit on a page suffering failed lines. Perfect pages are used when objects cannot be placed on a regular page, so the 3× reduction in demand for perfect pages by the two-page clustering algorithm reflects the strong defragmenting effect of failure clustering. The xalan benchmark makes very heavy use of perfect pages and consequently sees a huge advantage in two-page push at 10% failure rate. The counter-intuitive lowering of overhead with rising failure rate in xalan with one-page push is due to the effect of heap compensation, as it was with lusearch (Section 6.1).

Figure 9(b) also shows an interesting property of the two page clustering. The demand for clean pages is very robust to failure rates even as high as 50%. As long as a two-page region has a failure rate less than 50%, the clustering will yield at least one perfect page out of every two. Once the failure rate in a two-page region exceeds 50%, clustering will only yield a free region that is smaller than a page, greatly reducing the stock of failure-free pages. Figure 10 shows that two benchmarks, pmd and jython, are very sensitive to this threshold, but the others degrade gradually.

7. Discussion

This section discusses implications of our work.

7.1 Managed Language Incentives

Managed languages offer a layer of memory abstraction, which this paper exploits to attain fault tolerance transparently and which native languages lack. Our work thus creates an incentive to use managed languages for future memory technologies. Native applications will need to rely on the operating system to shield them from dynamic failures, losing an entire PCM page when the first line on the page has failed. Modifying a free-list allocator to handle static failures is possible. However, it is more complex than the solution offered by Immix. Without clustering, it would be even more complex. An additional consideration for native runtimes is that because C applications are unsafe they may generate illegal reads and writes to failed memory, which the OS will need to handle. Without support for pages with failures in a native runtime, native applications will have access to fewer and fewer pages as memory fails, making managed applications more attractive for PCM memory.

7.2 Wear Leveling Considered Harmful

Wear leveling strives to uniformly spread writes and wear all memory equally. However, the result is that failures end up being evenly spread out through memory, causing memory fragmentation. Our results show that clustering mitigates fragmentation, but only to a certain extent. We argue that evenly wearing memory is not the best wear management strategy because of the fragmentation it causes. Concentrating writes into certain regions of memory may reduce total system fragmentation and the effect of failures. We intend to investigate smarter wear management strategies in future work.

7.3 Balanced Hardware Clustering

Our results show that managed runtimes experience performance improvements with multiple-page regions because this keeps entire working pages available longer, making allocating medium and large objects less challenging. Native applications need entire working pages, so we expect them to benefit from this as well.

It may seem beneficial to use larger regions. Although initially a higher number of pages are logically intact (e.g., three in four-page region), these cases quickly degenerate to the two-page case. When failures reach roughly 25%, two of the four pages will be necessary to remap all failures in the region. Moreover, large regions may create additional complexity such as failure map cache pressure.

7.4 Possible Mitigation of Memory Fabrication Issues

As DRAM cells get smaller and closer to the atomic scale, precise manufacturing of these memory cells gets more challenging and fabrication variation causes cells to have wide ranges of sizes and electron-storage capacities. Manufacturing issues may cause certain cells to be 'born dead', i.e., not capable of holding any charge after fabrication. Similar issues may afflict other memory technologies. DRAM manufacturers today have on-chip provisions to replace a small number of failed cells before memory parts are shipped, but scaling this solution to a higher number of failures while keeping overheads low may be very challenging. This issue may cause low yields, i.e., only being able to ship the small portion of fabricated chips that have low number of failures, which increases costs and disrupts the memory industry business model. Our solution makes memory chips with arbitrary numbers of failures useful, without incurring high area or cost overhead. Instead of throwing the chips with high failures rates away, manufacturers can 'bin' them into classes. For example, the higher the number of failures, the cheaper the memory. This process is akin what processor manufacturers do today with chip frequency and power dissipation, which is also caused by process variation.

8. Conclusions

This paper explores an example of Rattner's "self-aware systems" [20], in which hardware and software cooperate to accomplish a shared system goal, in this case, system endurance. As hardware resources are gradually used up and exhausted, the OS and runtime come to the rescue to tolerate visible failures.

This work proposes a cooperative hardware/software system with low hardware and software complexity to mitigate failures in wearable memories. The OS and runtime coordinate to recover data from failed lines and migrate objects affected by failures. We use the memory abstraction provided by a managed runtime to handle failures transparently, and exploit the Immix garbage collector's capacity to efficiently skip over unusable holes. Even without hardware clustering, the runtime correctly handles failures, but overhead is 17% with 10% failed memory and 33% with 50% failed memory. When the hardware assists by clustering failures, software overhead reduces quite significantly to 3.9% with 10% failed memory and 12.4% with 50% failed memory. This paper is the first to observe that hardware and software cooperation has the potential to achieve substantially better failure tolerance with increasing numbers of hardware failures.

9. Acknowledgments

We thank Burton Smith for initial failure map implementation ideas, and our anonymous reviewers for their valuable feedback.

References

[1] D. Bacon, P. Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compiler, and Tool Support for Embedded Systems*, pages 81–92, 2003.

[2] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, 2000.

[3] S. M. Blackburn and K. S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 22–32, 2008.

[4] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *Proceedings of the 2004 ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 25–36, 2004.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190, 2006.

[6] H.-J. Boehm. Conservative GC algorithmic overview. http://www.hpl.hp.com/personal/Hans_Boehm/gc/gcdescr.html.

[7] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, D. Burger, B. C. Lee, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Sumposium on Operating Systems Principles*, pages 133–146, 2009.

[8] A. Demmers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, and S. Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 261–269, 1990.

[9] *Avoiding server downtime from hardware errors in system memory with HP Memory Quarantine*. Hewlett-Packard Corporation.

[10] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: Improving program locality. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 69–80, 2004.

[11] E. Ipek, J. Condit, E. B. Nightingale, D. Burger, and T. Moscibroda. Dynamically replicated memory: Building reliable systems from nanoscale resistive memories. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–14, 2010.

[12] ITRS Working Group. ITRS report. Technical report, International Technology Roadmap for Semiconductors, 2011.

[13] Jikes RVM. *Compiler Replay*, Dec. 2011. http://jikesrvm.org/Experimental+Guidelines.

[14] D. Lea. A memory allocator. http://g.oswego.edu/dl/html/malloc.html.

[15] Micron Technology Inc. PCM-based MCP. http://www.micron.com/products/multichip-packages/pcm-based-mcp?source=mb.

[16] M. K. Qureshi. Pay-as-you-go: Low-overhead hard-error correction for phase change memories. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 318–328, 2011.

[17] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 14–23, 2009.

[18] M. K. Qureshi, V. Srinivasan, and J. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 24–33, 2009.

[19] S. Raoux, G. Burr, M. Breitwisch, C. Rettner, Y. Chen, R. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H. L. Lung, and C. Lam. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[20] J. Rattner. Extreme scale computing. Keynote Speech at the 39th International Symposium on Computer Architecture, 2012.

[21] J. B. Sartor, S. M. Blackburn, D. Frampton, M. Hirzel, and K. S. McKinley. Z-rays: Divide arrays and conquer speed and flexibility. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 471–482, 2010.

[22] S. Schechter, G. Loh, K. Strauss, and D. Burger. Use ECP, not ECC, for hard failures in resistive memories. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pages 141–152, 2010.

[23] N. H. Seong, D. H. Woo, V. Srinivasan, J. A. Rivers, and H.-H. S. Lee. SAFER: Stuck-at-fault error recovery for memories. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 115–124, 2010.

[24] X. Yang, S. M. Blackburn, D. Frampton, J. B. Sartor, and K. S. McKinley. Why nothing matters: The impact of zeroing. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 307–324, 2011.

[25] D. H. Yoon, N. Muralimanohar, J. Chang, P. Ranganathan, N. P. Jouppi, and M. Erez. FREE-p: Protecting non-volatile memory against both hard and soft errors. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture*, pages 466–477, 2011.

[26] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 14–23, 2009.