

A Study of the Viability of Exploiting Memory Content Similarity to Improve Resilience to Memory Errors

Scott Levy¹, Kurt B. Ferreira², Patrick G. Bridges¹, Aidan P. Thompson², and Christian Trott²

¹Department of Computer Science, University of New Mexico,
{slevy|bridges}@cs.unm.edu

²Sandia National Laboratories*, {kbferre|athomps|crtrott}@sandia.gov

Abstract

Building the next-generation of extreme-scale distributed systems will require overcoming several challenges related to system resilience. As the number of processors in these systems grow, the failure rate increases proportionally. One of the most common sources of failure in large-scale systems is memory. In this paper, we propose a novel runtime for transparently exploiting memory content similarity to improve system resilience by reducing the rate at which memory errors lead to node failure. We evaluate the viability of this approach by examining memory snapshots collected from eight HPC applications and two important HPC operating systems. Based on the characteristics of the similarity uncovered, we conclude that our proposed approach shows promise for addressing system resilience in large-scale systems.

*Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

1 Introduction

Building the next-generation of extreme-scale distributed systems will require overcoming several challenges related to system resilience. As we aggregate larger numbers of processors to construct more powerful systems, the rate at which failures occur increases proportionally [35]. As the rate of failures increases, more time is spent preparing for and recovering from failures and less time is spent doing useful work. This effect is especially pronounced in systems that employ traditional checkpoint/restart techniques, as the entire computation has to be rolled back each time a failure occurs [10, 13].

Memory-related errors are one of the most frequently observed sources of node failure in large-scale distributed systems [35]. Moreover, power concerns may exacerbate this problem as we consider deploying low voltage memory chips that are more prone to error [7].

Effective fault tolerance strategies in extreme-scale systems may also need to address hardening operating systems against memory failures [14]. If every region of memory is equally likely to experience an uncorrectable error, we would expect to see relatively few errors in kernel memory because it typically occupies a much smaller memory footprint than the application. However, recent evidence suggests that kernel memory may be more prone to memory errors than other regions of memory [20].

In this paper, we present a novel approach for using content similarity in the memory of HPC systems to improve resilience to uncorrectable memory errors. We then evaluate the viability of this method by examining: (a) the application memory of eight important and representative HPC workloads running on a Cray XE6 supercomputer; and (b) regions of kernel memory for two well-known operating systems used in HPC. By carefully considering the characteristics of these memory regions, we estimate the relative costs and benefits of this approach.

The remainder of this paper is organized as follows: in the next section, we describe the approach used for this similarity analysis. Then in Section 3, we describe our test methodology and platform used. Section 4 presents our memory similarity results for each of our HPC workloads. Also, this section further analyzes our representative applications to develop an understanding of the costs of

maintaining this similarity information throughout the lifetime of an application. In Section 5, we review related work in the area and place the contribution of our work in that context. Finally in Section 6, we summarize our results and outline future avenues of promising research.

2 Proposed Approach

ASC Sequoia Marquee Performance Codes [26]	AMG	A parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids [17].
	IRS	Implicit Radiation Solver. Solves the radiation transport equation by the flux-limited diffusion approximation using an implicit matrix solution [24].
DOE Production Applications	CTH	A multi-material, large deformation, strong shock wave, solid mechanics code [28]
	LAMMPS	Large-scale Atomic/Molecular Massively Parallel Simulator. A classical molecular dynamics simulator [33].
Mantevo Mini- Applications [18, 32]	HPCCG	Designed to mimic the finite element generation, assembly and solution for an unstructured grid problem.
	phdMesh	Parallel Heterogeneous Dynamic Mesh. An application designed to mimic the contact search applications in an explicit finite element application.
Miscellaneous Applications	SAMRAI	Structured Adaptive Mesh Refinement Application Infrastructure. Designed to enable the application of structured adaptive mesh refinement to large-scale multi-physics problems [25].
	Sweep3D	Solves a 1-group time-independent discrete ordinates (Sn) 3D cartesian (XYZ) geometry neutron transport problem [27].

Table 1: A brief summary of HPC applications used

We propose to exploit memory content similarity to allow applications to recover from uncorrectable DRAM ECC errors that would otherwise lead to application termination or node failure. The basic idea is that when a memory error occurs on a page that is similar to one or more other pages in the

address space of an application, we can use information about the page's similarity to reconstruct the contents of the damaged page without needing to terminate the affected application or restart it from a known good state. Our approach consists of two components: (a) classifying pages to identify memory content similarity; and (b) using the memory content similarity we identify to reconstruct pages that suffer memory errors.

2.1 Page Classification

We begin by placing each page in the address space of an application into one of four categories:

- **DUPLICATE PAGES** : pages whose contents exactly match one or more other pages and include at least one non-zero byte.
- **ZERO PAGES** : pages whose contents are entirely zero.
- **SIMILAR PAGES** : pages that (a) are not duplicate or zero pages; and (b) can be paired with at least one other page in application memory such that the difference between the two can be represented by a `cx_bsdifff` [36] patch that is smaller than a tunable threshold. The results in this paper were collected using a threshold of 1024 bytes.
- **UNIQUE PAGES** : pages that do not fall into any of the preceding three categories.

In practice, we can treat zero pages as duplicate pages. If a memory error occurs on a zero page, reconstruction of the damaged page is straightforward. However on some systems, zero pages may be an artifact of memory allocation and may not represent memory that is actually being used. But if zero pages represent unused memory, they cannot be the source of memory errors. Therefore, zero pages can only increase the protective effect of our approach. Nonetheless, because we cannot determine which zero pages are actually used, our analysis distinguishes between zero pages and duplicate pages. The result is a possible underestimate of the protective benefit of our approach.

2.2 Page Reconstruction

When an uncorrectable ECC error is detected in an x86 system, the memory controller raises a Machine Check Exception (MCE) in the processor. The consequences of raising an MCE vary by operating system. Recent versions of Linux attempt to minimize the impact of an MCE by adopting simple recovery strategies. For example, in the event that the memory is unmapped,¹ the hardware page is poisoned and no other action is required. In the event that none of its recovery strategies is successful, Linux poisons the hardware page and kills all of the processes that had the faulted page mapped into their address space [22]. In other operating systems (e.g., the Kitten lightweight kernel [34], older versions of Linux), raising an MCE crashes a node, forcing the CPU to a halt state.

For each duplicate or similar page, we maintain a description of its *reference page(s)*: the set of pages in the system that are either duplicated by or similar to the page under consideration. In the case of similar pages, we also store the appropriate patch data. Because the patches generated by `cx_bsdifff` are not symmetric, *every* similar page requires its own patch data. Each time a duplicate or similar page is written to, we no longer know its relationship to its reference pages. As a result, we need to update our metadata to reflect the fact that the page must now be treated as unique. To determine whether these pages are still duplicate or similar after they have been written to, we would periodically compare each member in the set of unique pages (which includes similar and duplicate pages that have recently been written to) to other pages in the application’s memory. This would also allow us to identify new similar and duplicate pages from the set of unique pages.

When a memory error occurs on a similar or duplicate page, we can use the metadata that we have collected to reconstruct the faulted page. Reconstructing duplicate pages is straightforward. We simply restore the contents of the damaged page from the contents of one of its reference pages. For similar pages, the process is only modestly more complex. We reconstruct the damaged page by applying a patch to one of its reference pages.

¹This might happen if, for example, the MCE was raised by a memory scrubber. However, it is not clear that this is a common scenario [20].

3 Evaluation

To evaluate the viability of this approach, we considered the memory of several important HPC workloads and two key operating systems. By analyzing application and kernel memory, we were able to characterize memory content similarity in the system.

We examined memory in systems running the eight HPC workloads described in Table 1 using MPICH on 8 nodes of a Cray XE6 supercomputer. We used 8 processes on each node for a total of 64 MPI ranks. This set of applications is representative of several important workloads. In particular, three of these applications, AMG, IRS and LAMMPS, are taken from the ASC Sequoia Marquee Performance Codes: a set of codes assembled expressly to ensure that key workloads would perform well on the Sequoia supercomputer at Lawrence Livermore National Laboratory. Additionally, our set of applications includes two important U.S. Department of Energy (DOE) production applications: CTH and LAMMPS.

In addition, we examined similarity in the contents of kernel memory for two operating systems: Linux 2.6.37 (a full-weight kernel) and Kitten (a lightweight kernel) [34]. Although lightweight kernels have been shown to have superior performance characteristics [31], the generality and familiarity of full-weight kernels enable them to dominate today’s largest machines [3, 29, 30].

3.1 Data Collection

3.1.1 Application Memory

We built a library, `libmemstate`, to collect snapshots of the applications’ memory and linked it against each of the target applications. The MPI Profiling layer allows us to interpose `libmemstate` in all calls by the application to `MPI_Init` and `MPI_Finalize`. By intercepting the call to `MPI_Init`, `libmemstate` is able to take a snapshot of the application’s memory after initialization but before the application has started execution. To generate a snapshot of the application’s memory, `libmemstate` reads the

`/proc/<pid>/maps` file provided by Linux to gather information about the application’s address space. Based on the information it gathers, `libmemstate` writes a copy of the address space to stable storage.

After the initialization snapshot is complete, `libmemstate` sets a timed signal (`SIGALRM`) that allows it to take snapshots of memory periodically as the application runs. We collected the data in this paper by configuring `libmemstate` to capture a memory snapshot every 60 seconds of application execution time. The process is similar when the application calls `MPI_Finalize`. The MPI Profiling layer interposes a call to `libmemstate`. This allows `libmemstate` to take a finalization memory snapshot and disable its timer.

Each snapshot includes all of the application’s heap, stack and anonymous memory. We excluded memory-mapped files because for the applications that we considered, the majority of pages that correspond to memory-mapped files are mapped read-only. The most straightforward way to recover these pages is to re-read their contents from the backing store. As a result, our approach offers little additional protective benefit. While the pages that are backed by stable storage can be used as reference pages for other pages in application memory, we excluded these pages to simplify our analysis.

3.1.2 Kernel Memory

To collect data on kernel memory similarity, we used the checkpointing functionality of the Palacios Virtual Machine Monitor (VMM) [23] to capture the entire memory state of the guest periodically (once every 60 seconds for the data presented in this paper). Specifically, we used Palacios to capture snapshots of both Linux and Kitten guests running a single rank of each of six workloads.

After capturing the guest’s memory state, we required additional information to identify the regions of kernel memory within the guest memory footprint. We used different approaches for this purpose in Kitten and Linux. In Kitten, a region of low memory (by default, 64 MB) beginning at address 0 is reserved for kernel use. During the initial boot sequence, a very simple allocator (`bootmem`) is used to manage this memory. Near the end of the boot sequence, management of all unused kernel memory is transferred to a buddy allocator. By instrumenting the buddy allocator, we were able to

determine which pages of kernel memory were in use at any given instant. This allowed us to extract the relevant portions of the guest’s memory footprint. Although this approach fails to capture the memory allocated by the `bootmem` allocator, it does allow us to identify all of the memory allocated after the kernel memory subsystem has been initialized.

In Linux, every page of physical memory is represented by an instance of `struct page`. The `flags` field within each of these structures allowed us to determine characteristics about the page. In particular, pages that are managed by a slab allocator have the `PG_slab` bit set. Based on this structure, we built a kernel module that allowed us to traverse physical memory and determine which pages belonged to a slab cache.² Although this approach does not capture all of kernel memory (and captures some memory that may not be in active use), it does capture all of the memory allocated by `kmalloc`. Moreover, given the complexity of memory allocation in the Linux kernel, this is a relatively straightforward approach that allows us to approximate the similarity characteristics of kernel memory.

3.2 Data Analysis

We analyzed the set of memory snapshots that we collected of the applications and kernels offline. For each snapshot, we walked through the virtual address space from low addresses to high, categorizing each page of memory into one of the four categories described above: (a) duplicate; (b) similar; (c) zero; or (d) unique.

3.2.1 Duplicate Pages

Naively, identifying duplicate pages is a $O(n^2)$ operation. To reduce this cost, we compute the MD5 sum of each page and use it as the key of a hash table. Each collision represents a duplicate page. Although it is conceivable that two or more different pages could yield the same MD5 sum, we assume that for the applications that we considered the contents of application memory are not adversarial.

²Although the SLAB allocator has been largely replaced by the more efficient SLUB allocator [9], the “slab” nomenclature still predominates.

As a result, we can use an analogy to the birthday problem [19] to show that the likelihood of such an event is exceedingly small (i.e., $\approx 10^{-14}$) even for large memory snapshots [39].

3.2.2 Similar Pages

As with identifying duplicate pages, the naive approach to identifying similar pages is an $O(n^2)$ operation. To mitigate this cost, we use an approach inspired by the Difference Engine [16]. Instead of computing patches between every pair of pages, we attempt to identify a tractably small set of pages for each candidate page that are likely to be similar to it.

During initialization, we randomly choose four locations in a 4kB page of memory. For each page that we examine, we collect one 128-byte block at each of these locations. Each of these blocks is used as a *signature* of the page contents.

As we examine each candidate page in the address space of an application, we identify pages that match one or more of the candidate page’s signatures. In the event that more than one page matches a single signature we choose the page nearest to the candidate page. This approach identifies up to four pages that may be similar to the current candidate page. In addition to these pages, we also consider the page that occupies the next lowest virtual address in use in the application’s address space. In all, this approach identifies as many as five pages that are likely to be similar to the candidate page.

We then compute a patch between the current candidate page and each member of the set of likely similar pages. If any patch is smaller than a threshold, in this case 1024 bytes, we mark the current candidate page as similar. Because `cx_bsdifff` does not generate symmetric patches, observing a single patch that falls below our threshold is sufficient to categorize only a single page as similar. Therefore, we also compute the reciprocal patch of each of the pages in the set of likely similar pages to determine whether any of them should also be marked as similar.

This is a statistical, heuristic approach (*cf.* [16]). Although there may be methods that would yield greater numbers of similar pages by generating smaller patches, the fraction of similar pages we identify using this approach is a lower bound on the total number of similar pages in application memory.

3.2.3 Zero Pages

Identifying zero pages is a straightforward process that leverages the process of identifying duplicate pages. Initially, we make no effort to distinguish zero pages from any other page; we insert them into the duplicates hash table as we would any other page. By precomputing the MD5 sum of a 4kB zero page, we can then identify the zero pages as the set of pages that were stored in the duplicates hash table using the zero page MD5 sum as a key.

3.2.4 Unique Pages

Unique pages are those pages that fall outside of the criteria for the preceding three categories. However, we note that this is not a rigid definition; it is highly dependent on our choice of patch size threshold. In particular, increases to the patch size threshold will increase the number of similar pages and decrease the number of unique pages. With a sufficiently large patch size threshold, we could, in principle, transform all of the unique pages into similar pages. In subsequent sections, we examine the tradeoffs involved in changing the patch size threshold.

3.3 Repeatability

Non-determinism exists in our methods for collecting and analyzing data. With respect to data analysis, the source of non-determinism is explicit: as described above, we randomly choose the locations of four signatures. To estimate the variation introduced by this approach, we ran our analysis ten times on the memory snapshots collected for LAMMPS, randomly choosing the four signature locations each time. The number of similar pages varied by less than 0.23% across all of the snapshots (excluding the initialization and finalization snapshots) we collected.

With respect to data collection, the timers we use to determine the interval between memory snapshots are not precise. As a result, from run to run we cannot be sure that the snapshots are taken precisely relative to the application's progress. Therefore, it is unlikely that any two sequences

of memory snapshots will agree on the exact contents of memory at any given time. Moreover, there may be some variability in the layout of each application’s address space that may effect the content of the pages in an application’s memory.

To evaluate the variability in our data collection mechanism, we collected memory snapshots for ten separate runs of LAMMPS. We then categorized the pages for each sequence of memory snapshots. To control for the variability introduced by our analysis scripts, we fixed the locations of the four signatures used in our similarity detection algorithm. We observed that the percentage of similar pages varied by up to 27%, ranging from 15.6% to 20.0% of application memory.

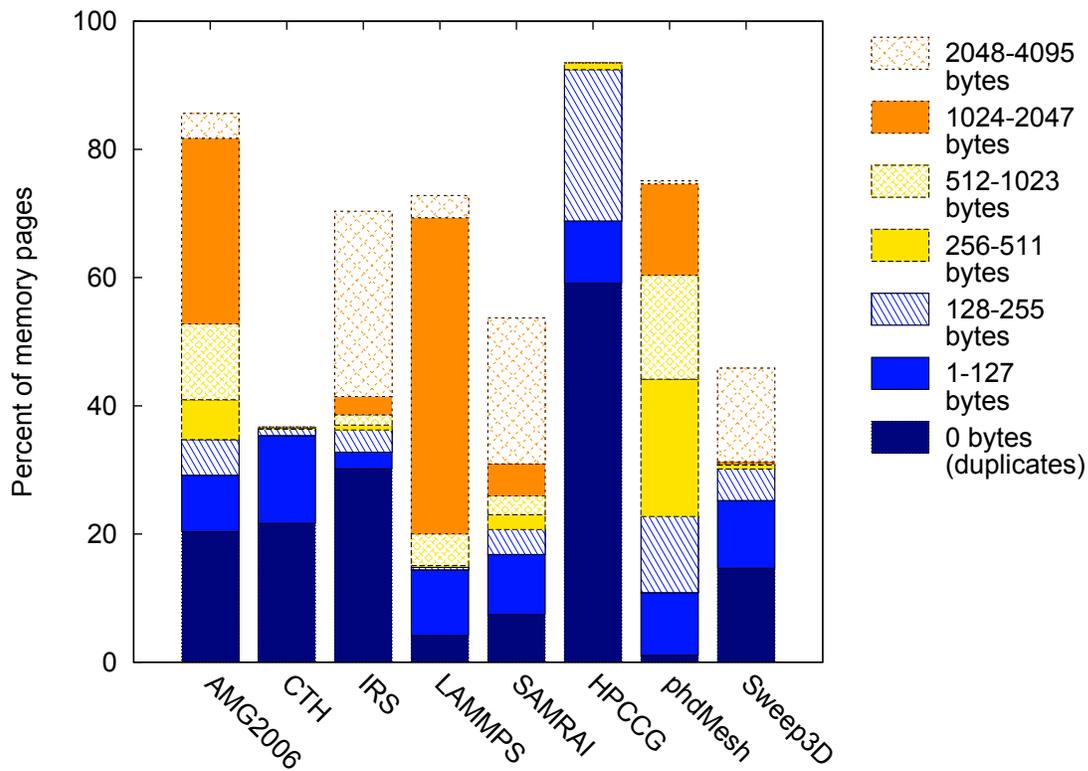


Figure 1: The percent of similar pages for rank 0 as a function of patch size threshold. A patch size of 4096 (a redundant copy of page with no actual similarity) will take all applications to 100%

Figure 1 demonstrates one possible source of this variation. For LAMMPS, a significant majority of the patches are between 1024 and 2047 bytes in size, just larger than our patch threshold of 1024 bytes. As a result, small changes in the content of a page (e.g., because the memory snapshots across executions are not synchronized relative to execution time) have the potential to cause many pages to be recategorized.

4 Results

4.1 Application Memory

In this section, we present the results of our examination of application memory. The goal of our examination is to develop an understanding of the potential benefits and costs of our proposed approach.

4.1.1 Benefits

The principal benefits of our approach will be expressed in terms of the degree and extent to which we are able to protect application memory against uncorrectable errors. To characterize this benefit, we examine the prevalence of similar and duplicate pages in application memory. In practice, we can also protect zero pages. However, because zero pages may be an artifact of memory allocation or other platform features, we exclude them from our consideration of our approach’s potential benefits.

Overview Figure 2 presents the fraction of each application’s address space that falls into each of the four categories described above. Excluding the initialization and finalization snapshots, this figure presents the results for the memory snapshot for rank 0 that contains the smallest fraction of similar and duplicate pages.

The first observation we make is that the memory of all of the applications is comprised of a significant fraction of pages that can be protected with our technique. For five of the applications (AMG, IRS, CTH, HPCCG and phdMesh), more than than 35% of the pages in their memory are

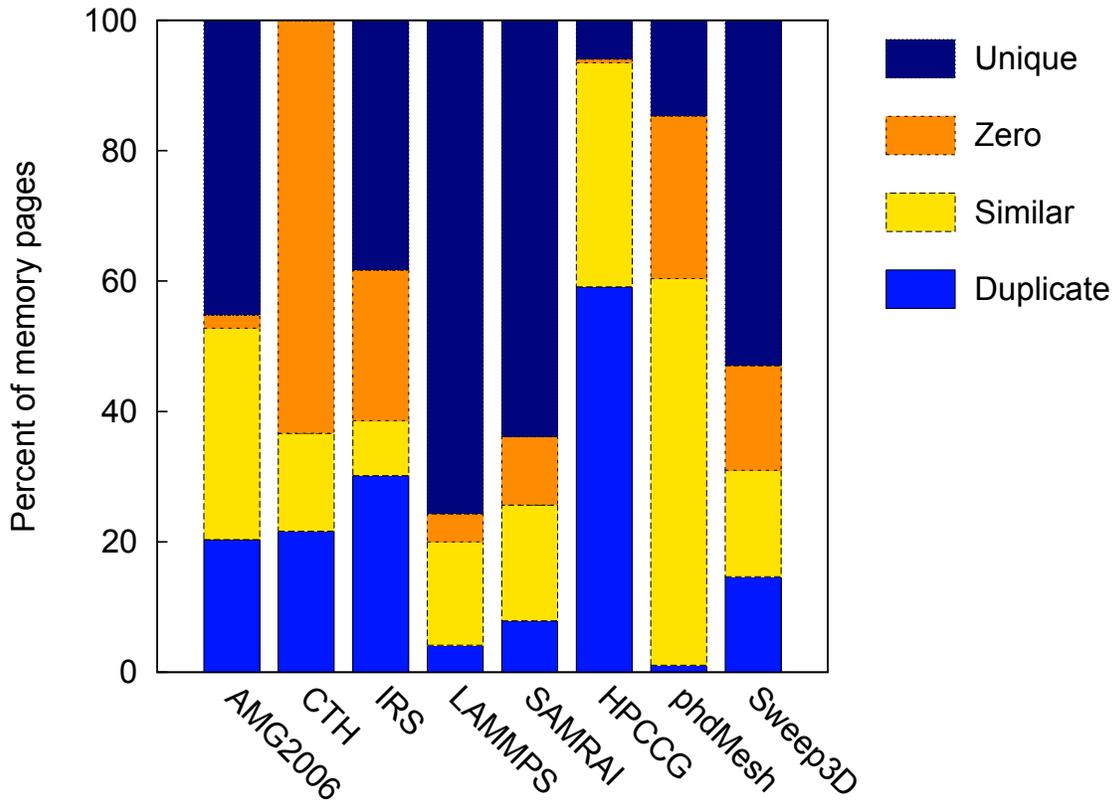


Figure 2: Page categorization within Rank 0 for each application. As discussed in Section 3.1, we collected memory snapshots every 60 seconds of application execution time, for a total of 7-11 snapshots per workload. This data represents the page categorization for the memory snapshot that contained the smallest fraction of similar and duplicate pages.

similar or duplicate pages. In no case do similar and duplicate pages comprise less than 20% of application memory.

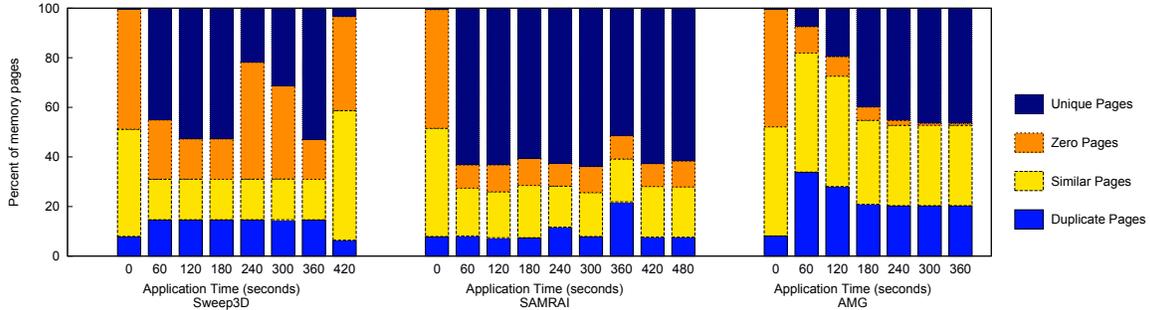


Figure 3: Three temporal behaviors observed in all our applications considering only similar and duplicate pages; Stable, the most commonly observed pattern, represented here by Sweep3D but also observed with LAMMPS, IRS, HPCCG, and phdMesh; Noisy, represented here by SAMRAI, but also observed with CTH; and, Dynamic, observed with AMG.

Temporal Behavior We next look at the behavior of memory content similarity over time. For the applications that we considered, we observe three distinct temporal trends in the fraction of similar and duplicate pages: (a) Stable; (b) Noisy; and (c) Dynamic. Examples of the behavior that is characteristic of each of these categories are shown in Figure 3. LAMMPS, IRS, HPCCG, phdMesh and Sweep3D constitute the temporally stable category. Excluding the initialization and finalization snapshots, the virtual address space of each these applications includes a stable fraction of duplicate and similar pages. In contrast, the fraction of similar and duplicate pages in the memory of CTH and SAMRAI is more erratic. They show significant fluctuations in the number of duplicate and similar pages in their virtual address spaces over the lifetime of the application. For example, the number of duplicate pages in SAMRAI spikes twice during this particular run; at one point the number of duplicate pages nearly triples. CTH exhibits similar, but less pronounced, noisy behavior. Early in its execution, the number of duplicate pages drops by more than 10%. Although the snapshots of

these applications captured only a handful of deviations, it suggests that the memory contents of these applications may be more dynamic and unpredictable than the other applications we considered. Lastly, AMG memory exhibits significantly different behavior in this regard than any of the other applications. For roughly the first half of its execution, the fraction of duplicate and similar pages in the memory of AMG steadily decreases before stabilizing for the remainder of its run.³

To summarize, for the majority of the workloads tested, the fraction of similar and duplicate pages was temporally stable. Only in SAMRAI, CTH, and AMG did the extent of similarity change significantly over the lifetime of the application. Because the protective benefit of our approach is highly dependent on the fraction of similar and duplicate pages in application memory, these data also suggest that for many applications the protective benefit of our proposed approach will be stable over the lifetime of the application.

NUMA We ran all of our tests of application memory on a Cray XE6 system. Because each of the XE6 compute nodes uses a NUMA architecture, we may be able to increase similarity by considering memory across processes.

Each compute node of the XE6 contains two 8-core AMD Opteron Magny-Cours processors. Each Magny-Cours processor is divided into two NUMA domains. Each NUMA domain is comprised of four cores [37]. We used the default MPICH layout method which results in SMP-style placement of MPI ranks. Based on this architecture, we were able to group our memory snapshots by rank to effectively examine content similarity within a NUMA domain for each application. The results of considering memory across a NUMA domain are shown in Table 2.

Expanding the scope of memory significantly increased the number of duplicate pages in memory of most of the applications we considered. For example, in LAMMPS, the number of duplicates increased by 148.6%. However, the number of similar pages decreased by nearly an equal amount for every

³We also observe that AMG is the only application that allocates significant quantities of memory after MPI initialization.

Application	Rank 0-3			NUMA Domain			Δ	Δ	% Increase
	# Similar	# Duplicate	Total	# Similar	# Duplicate	Total	Similar	Duplicate	Total
AMG2006	269748	185119	454867	222675	234162	456837	-47073	49043	0.4 %
CTH	27688	40507	68195	4691	63583	68274	-22997	23076	0.1 %
IRS	15085	55235	70320	11210	59320	70530	-3875	4085	0.3 %
LAMMPS	57922	14299	72221	36770	35541	72311	-21152	21242	0.1 %
SAMRAI	7841	4003	11844	4451	7437	11888	-3390	3434	0.4 %
HPCCG	297155	557443	854598	76327	778302	854629	-220828	220859	0.0 %
phdMesh	192590	8005	200595	188845	13921	202766	-3745	5916	1.1 %
Sweep3D	3748	3376	7124	965	6183	7148	-2783	2807	0.3 %

Table 2: Effect of considering the nodes in a single NUMA domain collectively. Although the number of duplicate pages increases significantly when all of the application memory in a NUMA domain is considered, these gains are almost entirely offset by reductions in the number of similar pages.

application we considered. As a result, processing the memory in a NUMA domain collectively yielded very modest increases in the total fraction of similar and duplicate pages.

The result is that, for many applications, there may be little incentive to collectively consider application memory within a NUMA domain. Nonetheless, because the cost of computing and storing metadata is higher for similar pages than for duplicate pages, there is a tradeoff to be made between local, similar pages and remote, duplicate pages.

Input Effects In addition to variations among applications, the input description for each application has the potential to impact the extent of content similarity. To examine the effect of changing inputs, we examined the memory of CTH and LAMMPS for several different inputs.

For CTH, we considered two inputs: (a) a model of the detonation of a conical explosive charge; and (b) a model of a fragmenting pipe. All preceding results for CTH presented in this paper were obtained using the conical charge input.

Figure 4(a) shows the fraction of similar and duplicate pages for each input. Each colored box represents the average fraction of similar or duplicate pages over the lifetime of the application (excluding the initialization and finalization snapshots). The error bars represent the minimum and maximum fraction of each category observed over the run. For the fragmenting pipe input, we observed substan-

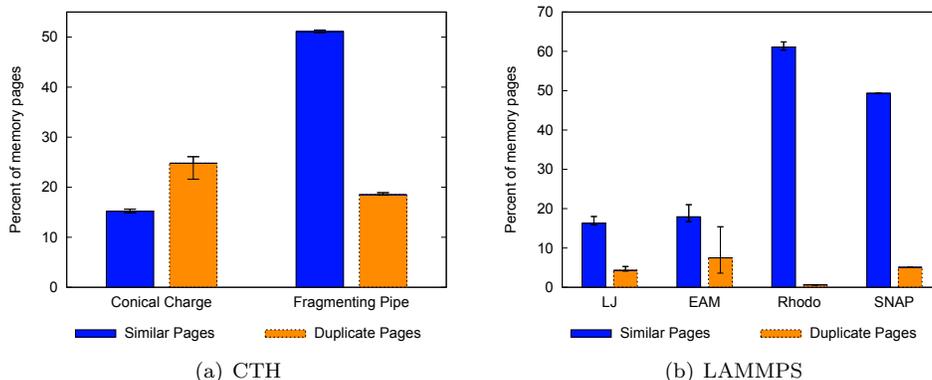


Figure 4: The effect of using different inputs on the fraction of similar and duplicate pages observed in application memory

tially higher percentage of similar pages than for the conical charge input. The relative frequency of similar and duplicate pages is also noticeably different between the two inputs. For the conical charge input, there are 60% more duplicate pages than similar pages whereas for the fragmenting pipe input, there are less than half as many duplicate pages as similar pages.

For LAMMPS, we considered four potentials as input: (a) Lennard-Jones (LJ); (b) Embedded Atom Model (EAM); (c) Rhodopsin (Rhodo) protein; and (d) SNAP⁴. All preceding results for LAMMPS presented in this paper were obtained using the LJ potential.

Figure 4(b) shows that the percentage of similar pages in the SNAP and Rhodo potentials is nearly twice as large as for the LJ and EAM potentials. Additionally, the fraction of duplicate pages is substantially lower for the Rhodo input than for the other three inputs.

These results illustrate that content similarity varies not only across applications but also across inputs to a single application.

⁴SNAP is a computationally intensive potential that uses the same kernel as the GAP potential [2].

4.1.2 Costs

Due to the metadata required, our proposed approach imposes two principal costs: (a) temporal costs; and (b) storage costs. The temporal costs include the number of CPU cycles that are taken from the application for metadata maintenance. The need for metadata maintenance is driven by how often the contents of similar and duplicate pages change. The storage costs include the number of bytes required to store the necessary metadata. The amount of storage required is largely dependent on the size of the patches for each of the similar pages.

Modification Behavior The time required to maintain the metadata necessary to make this approach work will be deducted from the time that the application would otherwise run. The magnitude of this temporal overhead will depend largely on the frequency with which similar and duplicate pages are modified. Each time a similar or duplicate page is modified, we no longer know the relationship between the page and its reference page(s). As a result, we need to update our metadata to account for this change. The more rapidly that similar and duplicate pages change, the higher the temporal overhead of managing metadata will be.

Application	Changed 1+ Times	Changed 1 Time	Changed 2 Times	Changed 3 Times	Changed 4+ Times
AMG2006	20.8 %	9.9 %	5.4 %	1.7 %	3.8 %
CTH	38.9 %	6.9 %	3.5 %	13.7 %	14.8 %
IRS	32.8 %	18.3 %	0.2 %	0.0 %	14.3 %
LAMMPS	37.6 %	0.5 %	0.6 %	0.5 %	36.0 %
SAMRAI	79.5 %	13.6 %	7.7 %	32.0 %	26.3 %
HPCCG	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %
phdMesh	21.6 %	6.2 %	1.9 %	0.5 %	13.0 %
Sweep3D	4.1 %	1.7 %	0.7 %	0.0 %	1.8 %

Table 3: Modification behavior of the pages in the memory of Rank 0 that are ever categorized as similar or duplicate.

To estimate how frequently similar and duplicate pages change, we compared the memory contents across the sequence of snapshots we collected for each application. By hashing each page, we were able to determine whether a given page in the application’s virtual address space changed from one snapshot to the next.⁵ Table 3 shows the modification behavior for all of the pages in application memory that are *ever* classified as duplicate or similar.

The data in this table suggests that for most applications, a substantial majority of the similar and duplicate pages are either read-only/read-mostly or are written to without being modified [12]. For five of the eight applications (AMG, IRS, HPCCG, phdMesh and Sweep3D), more than 84% of the similar and duplicate pages are modified either once or not at all.

The modification behavior of HPCCG is particularly striking; a vanishingly small percentage of its similar or duplicate pages are ever modified. A more detailed examination of HPCCG’s application memory shown in Figure 5 reveals that the range of the application’s address space occupied by similar and duplicate pages is almost entirely disjoint from the range occupied by modified pages.

Similar and duplicate pages are confined to the low end of the virtual address space and modified pages occupy the high virtual addresses. We speculate that because HPCCG is a conjugate gradient solver, the low end of the virtual address space contains the sparse matrix that is provided as input (and is never modified) and the high virtual addresses contain the solution vector that is refined on each iteration.

Although the heat maps for HPCCG appear to tell a coherent story about the source of the similarity, the same does not hold for the other applications we considered. In general, the pattern of similarity within the application’s memory appears to reveal little about the source of the similarity. For example, Figure 6 shows the similarity and modification heat maps for CTH. Although these figures are not without structure, reasoning about the source of the similarity based on these plots is challenging. The heat maps for the other six applications (IRS, Sweep3D, AMG, SAMRAI, and

⁵This approach underrepresents the frequency of page modifications because it does not necessarily account for multiple modifications if they occur between memory snapshots.

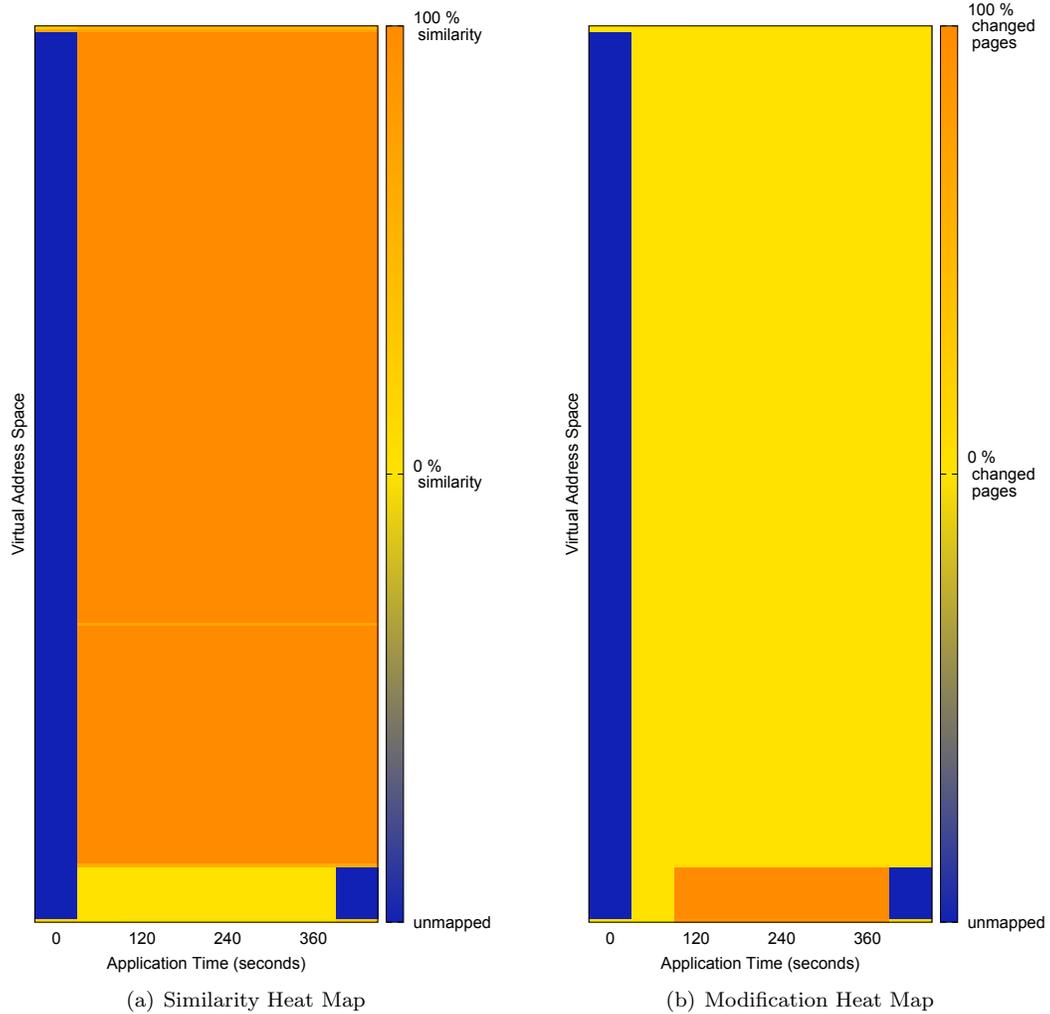


Figure 5: HPCCG Rank 0 address space Similarity and modification heat maps. Due to HPCCG’s computation pattern, the address space occupied by similar and duplicate pages is almost entirely disjoint from the range occupied by modified pages

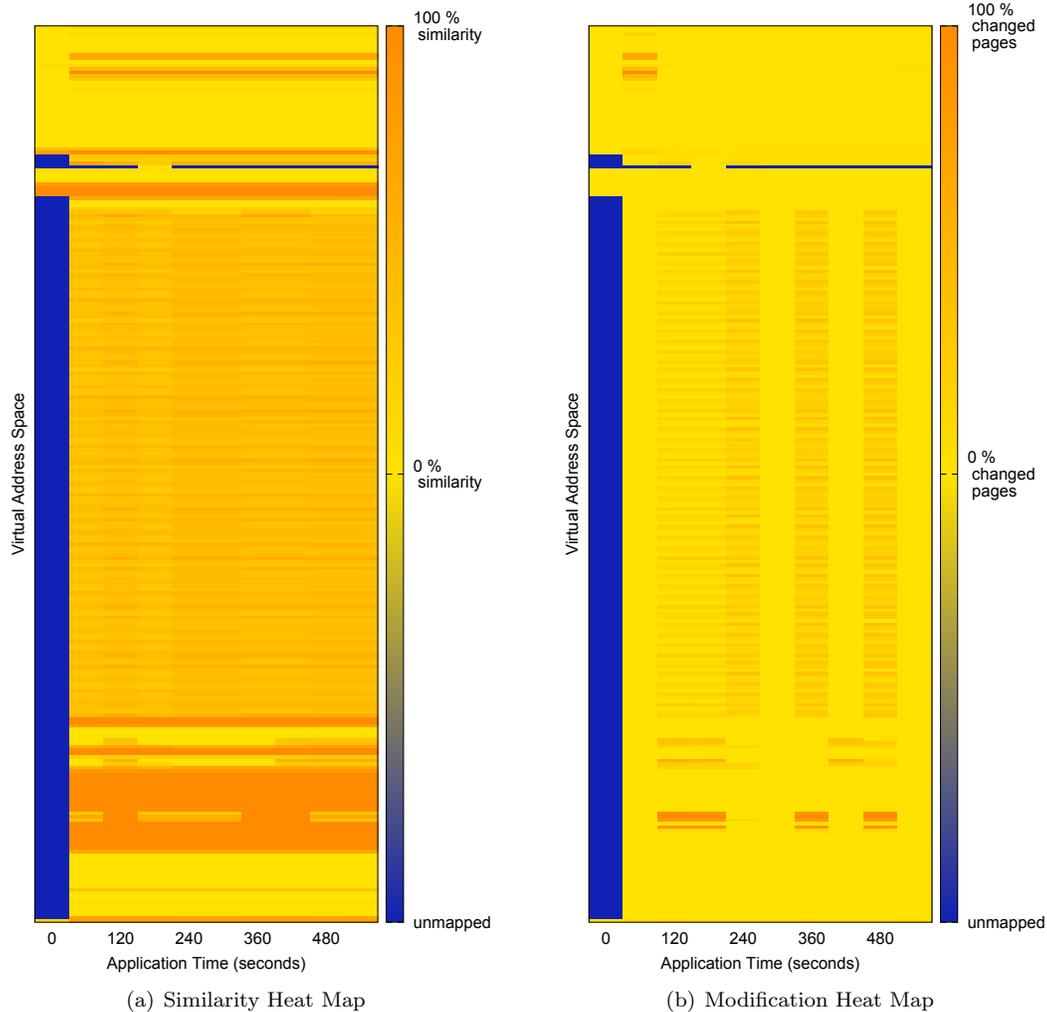


Figure 6: Address space similarity and modification heat maps for CTH. These figures show the evolution of the memory characteristics of rank 0 over time. Unlike HPCCG, reasoning about the relationship between these figures and application data structures is challenging. The heat maps for LAMMPS, IRS, Sweep3D, AMG, SAMRAI, and phdMesh are similarly difficult to reason about. Identifying the source of similarity will require significant application expertise.

phdMesh) are similarly difficult to reason about. This is due in large part to the difficulty of reasoning about the relationship between application data structures and similarity. We also note that HPCCG is a mini-application while many of the other workloads that we considered are more complete (and complex) applications.

Despite these promising results, there are applications that frequently modify similar and duplicate pages. For example, unlike the other applications that we considered, a majority of the similar and duplicate pages in the memory of SAMRAI are modified at least once; more than half are modified three or more times.

Taken as a whole, these results indicate that similar and duplicate pages are comprised largely of read-only and read-mostly data. As a result, the metadata associated with these pages need only be infrequently updated. This evidence suggests that, for many applications, the overhead of our proposed approach will be manageable and commensurate with its protective effect.

Patch Size Threshold The patch size threshold represents a trade-off between the number of pages that are similar and the quantity of metadata that must be maintained. A threshold of 1024 bytes strikes a conservative balance between maximizing similarity and minimizing metadata.

For six of the eight applications that we considered, the metadata associated with 1024-byte threshold would occupy less than 1.5% of the application memory. The metadata for AMG and phdMesh would occupy a slightly larger, but still modest, fraction (4.0% and 5.8%, respectively) of the application’s memory. By changing the patch size threshold, we can strike a different balance between the number of similar pages and the size of the associated metadata.

Figure 7 shows the fraction of similar and duplicate pages as a function of metadata size for each of the applications that we considered. The slope of the curves represents the ratio of cost to benefit. For applications like AMG, LAMMPS and phdMesh, we can extract significant similarity with a modest increase in metadata. In particular, for phdMesh, if we allow the metadata to occupy even a small fraction of application memory we see dramatic gains in the number of similar pages. IRS, Sweep3D

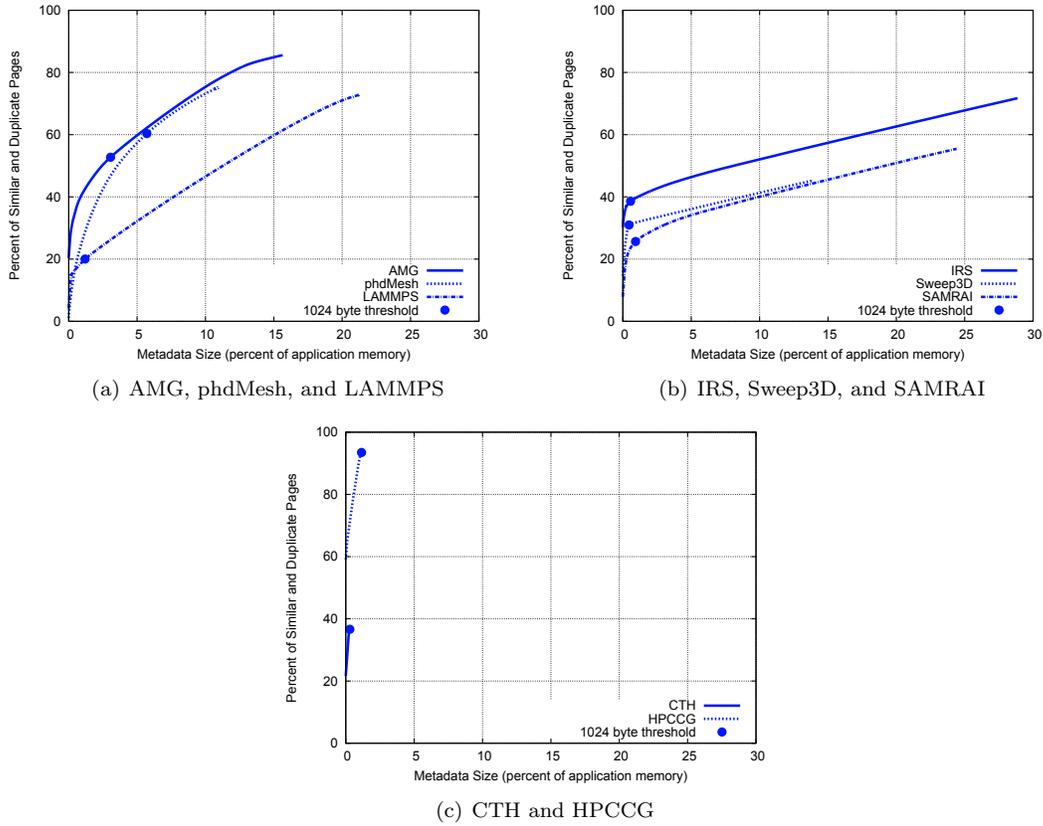


Figure 7: The percent of similar and duplicate pages as a function of metadata size. Figure 7(a) shows the three applications for which the cost of increasing the patch size is modest. Figure 7(b) shows the three applications for which increasing the patch size comes at a higher cost. Figure 7(c) shows the applications for which the cost of increasing the patch size is quite small. On each of the curves, a solid circle indicates the point on the curve that corresponds to a patch threshold of 1024 bytes.

and SAMRAI represent applications for which the benefits come at a higher cost. Finally, in the case of CTH and HPCCG, increasing the patch size increases the metadata by a very small amount. There are two principal reasons for this behavior. First, the patch sizes for these two applications happen to be quite small. As shown in Figure 1, most of the patches for these two applications are smaller than 256 bytes. Second, the memory of these applications are dominated by pages that are not suitable for

patching: zero pages for CTH; duplicate pages for HPCCG. Because there are a small number of small patches, increasing the patch size requires very little additional metadata for these two applications.

4.2 Kernel Memory

In this section, we examine similarity in kernel memory. Although the case for resilient operating systems is still emerging [14], we discuss why these results are promising for our proposed approach.

4.2.1 Similarity Overview

Figures 8(a) and 8(b) shows the composition of kernel memory for two important operating systems: Linux (a heavyweight OS) and Kitten (a lightweight OS). The data for each operating system represents the snapshot that exhibits the smallest extent of similarity. These data were collected while a user process was running one of six workloads. For Kitten, we considered all of the pages of kernel memory whose contents are ever managed by the buddy allocator. For Linux, we considered all of the kernel memory that is in a slab allocator at any point during the application’s execution.

The first observation we make is that both Linux and Kitten have a very large number of similar pages. Also, the kernel memory of both operating systems contains very few duplicate pages. This result is consistent with how the OS uses this memory; the majority of the state maintained by these OSs is comprised of table-based structures containing objects such as page table mappings. Given the nature of page tables, we would expect to find large numbers of similar pages in memory allocated for page table data structures. For x86 processors, each element in the page table hierarchy occupies a full 4kB page of memory even if only a handful of pages are mapped in the referenced region of virtual memory. As a result, page table data structures tend to be very sparse. Because the difference between any two sparse pages can be compactly represented, our approach will identify significant similarity in memory comprised of sparse pages. To empirically validate these observations, we instrumented Kitten’s buddy allocator to track the percentage of buddy-allocated memory that is used to store

page table data structures. For each of the six applications we considered, the minimum observed percentage of memory that is allocated by the buddy allocator for page tables is shown in Table 4. This data shows that page table data structures occupy the vast majority of buddy-allocated memory and thus must also be a significant source of the similarity observed in Figure 8(a).

We also observe that Linux has a much higher fraction of unique pages than Kitten. We speculate these unique pages comprise some portion of the Linux buffer cache. In contrast, Kitten lacks a kernel buffer cache and handles buffering in userspace memory. Finally, the fraction of similar pages in Linux kernel memory is largely unaffected by the application that is running. In contrast, the fraction of similar pages in Kitten kernel memory varies from application to application.

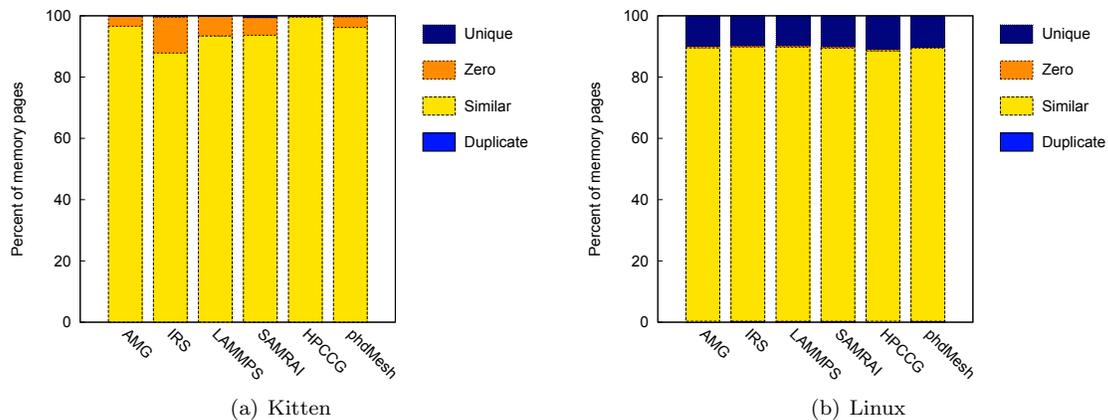


Figure 8: Page categorization within kernel memory for each kernel using a 1024 byte patch threshold. As discussed in Section 3.1, we collected memory snapshots every 60 seconds of application execution time, for a total of 5-6 snapshots per workload. Each bar represents the page categorization for the memory snapshot that contained the smallest fraction of similar and duplicate pages. In other words, these are the data that are least favorable to our proposed approach.

4.2.2 Patch Size Threshold

We now consider the tradeoff between patch size and memory overhead in these HPC operating systems. Figure 9 shows the fraction of similar and duplicate pages as a function of metadata size. The data

Application	Minimum Percentage of Buddy-Allocated Memory Used for Page Tables
AMG2006	93.7 %
IRS	85.3 %
LAMMPS	90.8 %
SAMRAI	90.9 %
HPCCG	97.0 %
phdMesh	93.7 %

Table 4: Minimum percentage of memory allocated by Kitten’s buddy allocator that is used for page table data structures over the lifetime of each of six workloads.

in this figure were collected during a run of HPCCG; similar results obtain for the other applications. The slope of these curves represents the ratio of cost to benefit. For Kitten, increasing the patch size results in a dramatic increase in the fraction of similar pages yet it requires only a very small increase in metadata size. For Linux, increasing the patch size comes at a greater (but still modest) cost. For both OSs, only a modest amount of metadata (less than 10%) is required to protect all of kernel memory using our proposed approach.

4.2.3 Modification Behavior

The cost of maintaining the metadata necessary to correct memory errors will depend, in part, on the rate at which similar and duplicate pages are modified. To examine the frequency of kernel memory modification, we again compared the contents of memory pages across the sequence of snapshots we collected. The results are shown in Table 5. For all of the application on both of the OSs, a significant majority of similar and duplicate pages are not modified during the application’s execution. For IRS running on Kitten, the rate of modification is substantially higher than for any other configuration. However, even in this case, less than one percent of the pages we considered were modified two or more times. These results are consistent with how these operating systems use memory; they construct tables

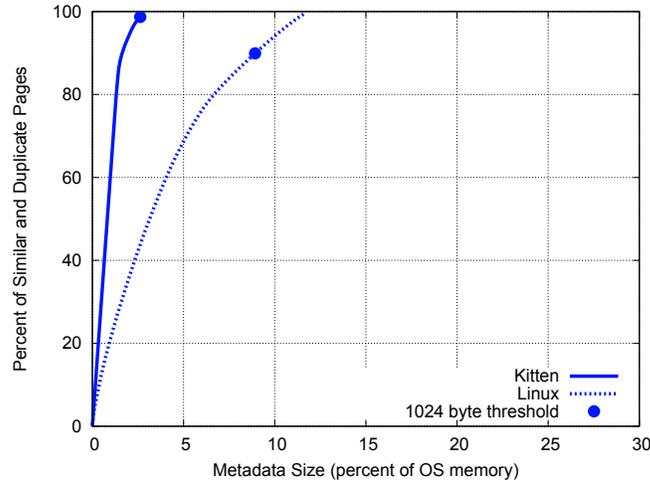


Figure 9: Fraction of similar and duplicate pages as a function of metadata size for Kitten and Linux running HPCCG. For each plot the data point corresponds to a patch size threshold of 1024 bytes.

that are written once and read many times. The infrequent modification of similar and duplicate pages in kernel memory suggests that the cost of metadata maintenance will be low.

The results in this section point to the potential of this novel technique to efficiently protect against uncorrectable memory errors in kernel memory. For both Linux and Kitten, significant similarity exists in regions of kernel memory. Additionally, similar and duplicate pages in kernel memory are infrequently modified and can be protected with small volumes of metadata.

5 Related Work

Memory content similarity has been explored for more than a decade. As a result, a significant body of relevant research has emerged. Although memory content similarity has been examined in several contexts, the preponderance of the relevant research has been in virtualization. The Disco VMM [6] included transparent memory sharing which exploited memory content similarity to reduce virtual

Operating System	Application	Changed 1+ Times	Changed 1 Time	Changed 2 Times	Changed 3 Times	Changed 4+ Times
Kitten	AMG2006	0.7 %	0.5 %	0.1 %	0.0 %	0.1 %
	IRS	25.4 %	24.8 %	0.1 %	0.0 %	0.4 %
	LAMMPS	0.5 %	0.2 %	0.2 %	0.0 %	0.1 %
	SAMRAI	1.6 %	0.7 %	0.4 %	0.1 %	0.3 %
	HPCCG	0.2 %	0.1 %	0.0 %	0.0 %	0.1 %
	phdMesh	1.5 %	1.5 %	0.0 %	0.0 %	0.0 %
Linux	AMG2006	2.2 %	1.7 %	0.1 %	0.0 %	0.4 %
	IRS	2.3 %	1.5 %	0.1 %	0.0 %	0.7 %
	LAMMPS	2.2 %	1.7 %	0.1 %	0.1 %	0.3 %
	SAMRAI	2.5 %	1.6 %	0.2 %	0.1 %	0.6 %
	HPCCG	1.6 %	1.0 %	0.0 %	0.0 %	0.6 %
	phdMesh	1.9 %	1.4 %	0.0 %	0.0 %	0.5 %

Table 5: Modification behavior of the pages in kernel memory that are ever categorized as similar or duplicate.

machine memory consumption. By intercepting disk requests that use DMA to transfer data into memory, the Disco VMM consolidated read-only pages (e.g., text segments of applications, read-only pages in the buffer cache) containing data from the disk across virtual machines. In some cases, this approach allowed the Disco VMM to significantly reduce memory consumption.

More recently, the VMware ESX server incorporated a broader approach to memory de-duplication [38]. Instead of intercepting disk requests, the server identified all pages in a virtual machine by their contents. When any two pages are found to have the same contents, the pages are consolidated using copy-on-write. Applying this approach to systems running as many as 10 identical VMs running the SPEC95 benchmark on Linux, the VMware ESX server is able to reduce memory consumption by nearly 60%.

Xia and Dinda have advocated broadening the scope of sharing in virtualization to consider intranode sharing [39]. To evaluate the feasibility of this approach, they examined the prevalence of duplicate pages within and across nodes running several HPC applications. For some workloads (notably HPCCG), they observed that significant inter- and intra-node sharing opportunities exist. Based

on these promising results, they proposed a Content-Sharing Detection System for exploiting intranode sharing in virtualized environments. Similarly, *SBLMalloc* has been used to demonstrate that memory consumption can be significantly reduced by consolidating duplicate pages in the application memory of several HPC applications [4].

Most memory de-duplication research has considered consolidating only duplicate pages. However, the Difference Engine [16] demonstrated that similar pages could also be consolidated. In this context, two pages are similar if the difference between them can be represented by an `xdelta` patch file that is smaller than 2kB.

In addition to virtualization, content duplication has been effectively exploited in other domains. In context of data storage, reducing storage requirements in primary and archival data storage applications by eliminating duplicate data blocks has been widely studied [40, 41]. Kernel Shared Memory (KSM) allows duplicate memory to be consolidated in Linux with or without virtualization [1].

A number of resilience techniques have been explored for HPC. Traditional checkpoint/restart [10, 11] is the most common approach. Asynchronous checkpointing [15, 21] and replication [13] have also been considered. In addition to these system-level approaches, algorithm-based techniques for enabling applications to withstand memory errors have been explored [5, 8]. In contrast, our approach will allow the system to transparently recover from memory errors without requiring application restart or detailed application knowledge.

6 Conclusion and Future Work

In this paper, we have described a novel approach for improving system resilience by exploiting similarities in system memory. We have also demonstrated the feasibility of this approach by presenting data indicating that significant similarity exists in several important HPC applications. We draw five specific conclusions from the data and analysis presented here.

- Significant similarity (greater than 35%) exists for several applications even with a conservative patch size threshold. Given the extent of memory content similarity, if we assume that memory errors are distributed uniformly over the virtual address space of an application, the approach we propose has the potential to reduce the rate of memory-induced application failure by a significant fraction.
- Most of the similarity and duplication comes from pages that are modified infrequently. This suggests that the temporal overhead of our proposed approach may be manageable relative to its protective benefit.
- For the applications that we considered, expanding the scope of the memory that we consider to include a NUMA domain provides a very modest improvement. This effect is due to the fact that the increase in duplicate pages is largely offset by a decrease in similar pages. Nonetheless, there may be circumstances in which we should choose local, similar pages over remote, duplicate pages. The costs and benefits of this trade-off will be explored more fully in our future work.
- Memory content similarity is not determined by the application alone. Even for a single application, the degree to which application memory is comprised of duplicate and similar pages varies significantly across inputs.
- Kernel memory in Linux and Kitten is comprised of a large fraction (greater than 85%) of similar pages. Moreover, the associated storage costs are modest; the metadata would occupy a small fraction (less than 10%) of memory.

While these results are promising, we do not yet have data showing the impact of this approach on application runtime. However, based on existing work in memory de-duplication [4, 16] we are optimistic that this overhead will be reasonable. For example, the application performance in systems using the Difference Engine, which also exploits page similarity at runtime, is within 7% of native [16].

Taken as a whole, these initial results suggest that using memory content similarity can be a very effective technique for correcting errors in application memory. As a result, we intend to pursue this idea further and to begin work on implementing a runtime that can, by exploiting memory content similarity, reduce the rate at which memory errors lead to node failure.

Acknowledgments

The authors gratefully acknowledge a number of associates from Sandia National Laboratories and university partners for their assistance in this work. First, we thank Kevin Pedretti at Sandia for his help and support understanding the Kitten lightweight kernel. Second, we thank Courtenay Vaughan at Sandia for his help in configuring the two CTH problems used in this work. We also thank Peter Dinda of Northwestern University and Jack Lange of the University of Pittsburgh for their help configuring our Palacios-based OS similarity test framework. Finally, we wish to thank Sandia's Laboratory Directed Research and Development (LDRD) office for their financial support of this work.

Authors' Biographies

Scott Levy is a Ph.D. student in the Computer Science Department at the University of New Mexico and an intern at Sandia National Laboratories. His research focuses on fault tolerance for next-generation HPC systems.

Kurt B. Ferreira is a senior member of Sandia's technical staff. He is an expert on system software and resilience/fault-tolerance methods for extreme-scale, massively parallel, distributed-memory, scientific computing systems. He has designed and developed innovative high-performance and resilient low-level system software for a number of HPC platforms, including the Cray Red Storm (XT3) machine at Sandia National Laboratories. His research interests include the design and construction of operating systems for massively parallel systems and innovative application- and system-level fault-tolerance

mechanisms for HPC.

Patrick G. Bridges is an associate professor in the Department of Computer Science at the University of New Mexico. His research interests include operating system and network protocol design, virtualization systems, and fault tolerance, particularly for high-performance computing systems.

Aidan P. Thompson is a member of the technical staff at Sandia National Laboratories, where he has worked for 18 years. He holds a Ph.D. in Chemical Engineering from the University of Pennsylvania. He is the main developer of the SNAP class of interatomic potentials, which use machine-learning techniques to achieve quantum-level accuracy in classical molecular dynamics (MD) simulations. In addition, he leads several efforts to apply large-scale MD simulations using advanced interatomic potentials running on advanced computer architectures, most notably the ReaxFF potential for reactive MD simulations of energetic materials.

Christian Trott is a research scientist in the Scalable Algorithms Group at Sandia National Laboratories. He received his PhD in theoretical physics at the University of Technology Ilmenau, Germany in 2011. Christian is now focused on enabling the development of performance portable scientific code for current and future many-core architectures.

References

- [1] A. Arcangeli, I. Eidus, and C. Wright. Increasing memory density by using KSM. In *Proceedings of the Linux Symposium, 2009, Montreal, Quebec*, pages 19–28, 2009.
- [2] A. Bartók, M. Payne, R. Kondor, and G. Csányi. Gaussian approximation potentials: The accuracy of quantum mechanics, without the electrons. *Physical Review Letters*, 104(13):136403, 2010.
- [3] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, P. Kogge, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems. [http://www.science.energy.gov/ascr/Research/CS/DARPA/exascale-hardware\(2008\).pdf](http://www.science.energy.gov/ascr/Research/CS/DARPA/exascale-hardware(2008).pdf), Sept. 2008.

-
- [4] S. Biswas, B. R. d. Supinski, M. Schulz, D. Franklin, T. Sherwood, and F. T. Chong. Exploiting data similarity to reduce memory footprints. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 152–163, Washington, DC, USA, 2011. IEEE Computer Society.
- [5] P. G. Bridges, M. Hoemmen, K. B. Ferreira, M. A. Heroux, P. Soltero, and R. Brightwell. Cooperative application/OS DRAM fault recovery. In *Euro-Par 2011: Parallel Processing Workshops*, pages 241–250. Springer, 2012.
- [6] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, Nov. 1997.
- [7] V. Chandra and R. Aitken. Impact of technology and voltage scaling on the soft error susceptibility in nanoscale CMOS. In *Defect and Fault Tolerance of VLSI Systems, 2008. IEEE International Symposium on*, DFTVS'08, pages 114–122. IEEE, 2008.
- [8] Z. Chen and J. Dongarra. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In *Proceedings of the 2006 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '06, April 2006.
- [9] Corbet. The SLUB allocator. <http://lwn.net/Articles/229984/>, Apr. 2007.
- [10] E. Elnozahy and J. Plank. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *Dependable and Secure Computing, IEEE Transactions on*, 1(2):97–108, 2004.
- [11] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, Sept. 2002.
- [12] K. Ferreira, R. Riesen, R. Brighwell, P. Bridges, and D. Arnold. Libhashckpt: Hash-based incremental checkpointing using GPUs. *Recent Advances in the Message Passing Interface*, pages 272–281, 2011.
- [13] K. Ferreira, R. Riesen, J. Stearley, J. H. L. III, R. Oldfield, K. Pedretti, P. Bridges, D. Arnold, and R. Brightwell. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage, and Analysis, (SC'11)*, Nov 2011.
- [14] K. B. Ferreira, K. Pedretti, R. Brightwell, P. G. Bridges, D. Fiala, and F. Mueller. Evaluating operating system vulnerability to memory errors. In *Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers*, page 11. ACM, 2012.
- [15] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello. Uncoordinated checkpointing without domino effect for send-deterministic message passing applications. In *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '11, May 2011.
- [16] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. *Commun. ACM*, 53(10):85–93, Oct. 2010.

- [17] V. Henson and U. Yang. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41(1):155–177, 2002.
- [18] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.
- [19] L. Holst. The general birthday problem. *Random Structures and Algorithms*, 6(2-3):201–208, 1995.
- [20] A. A. Hwang, I. A. Stefanovici, and B. Schroeder. Cosmic rays don’t strike twice: Understanding the nature of DRAM errors and the implications for system design. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’12, pages 111–122, New York, NY, USA, 2012. ACM.
- [21] D. B. Johnson and W. Zwaenepoel. Recovery in distributed systems using asynchronous message logging and checkpointing. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 171–181, 1988.
- [22] A. Kleen. Mcelog: Memory error handling in user space. In *Proceedings of Linux Kongress 2010*, Nuremburg, Germany, September 2010.
- [23] J. R. Lange, K. T. Pedretti, T. Hudson, P. A. Dinda, Z. Cui, L. Xia, P. G. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *Proceedings of the 2010 IEEE International Parallel and Distributed Processing Symposium*, IPDPS’10, pages 1–12, 2010.
- [24] Lawrence Livermore National Laboratories. IRS: Implicit Radiation Solver 1.4 Build Notes. https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/irs/irs.readme.html.
- [25] Lawrence Livermore National Laboratories. SAMRAI. <https://computation.llnl.gov/casc/SAMRAI/index.html>.
- [26] Lawrence Livermore National Laboratories. ASC Sequoia Benchmark Codes. <https://asc.llnl.gov/sequoia/benchmarks>, August 2009.
- [27] Los Alamos National Laboratories. Sweep3d. http://www.c3.lanl.gov/pal/software/sweep3d/sweep3d_readme.html, 1999.
- [28] J. McGlaun, S. Thompson, and M. Elrick. CTH: A three-dimensional shock wave physics code. *International Journal of Impact Engineering*, 10(1):351–360, 1990.
- [29] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. June 2013 — TOP500 Supercomputer Sites. <http://top500.org/lists/2013/06/>, June 2013.

- [30] K. Noyes. 94 Percent of the World's Top 500 Supercomputers Run Linux. <http://www.linux.com/news/enterprise/high-performance/147-high-performance/666669-94-percent-of-the-worlds-top-500-supercomputers-run-linux->, Nov. 2012.
- [31] R. Riesen, R. Brightwell, P. G. Bridges, T. Hudson, A. B. Maccabe, P. M. Widener, and K. Ferreira. Designing and implementing lightweight kernels for capability computing. *Concurrency and Computation: Practice and Experience*, 21(6):793–817, Apr. 2009.
- [32] Sandia National Laboratories. Mantevo. <http://software.sandia.gov/mantevo>.
- [33] Sandia National Laboratories. The LAMMPS molecular dynamics simulator. <http://lammps.sandia.gov>, April 2010.
- [34] Sandia National Laboratories. Kitten lightweight kernel. <https://software.sandia.gov/trac/kitten>, March 10 2012.
- [35] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, June 2006.
- [36] A. Tuininga. Cx_bsdiff. http://starship.python.net/crew/atuining/cx_bsdiff/index.html, February 2006.
- [37] C. Vaughan, M. Rajan, R. Barrett, D. Doerfler, and K. Pedretti. Investigating the impact of the Cielo Cray XE6 architecture on scientific application codes. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1831–1837. IEEE, 2011.
- [38] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, Dec. 2002.
- [39] L. Xia and P. A. Dinda. A case for tracking and exploiting inter-node and intra-node memory content sharing in virtualized large-scale parallel systems. In *Proceedings of the 6th International Workshop on Virtualization Technologies in Distributed Computing*, VTDC '12, pages 11–18, New York, NY, USA, 2012. ACM.
- [40] T. Yang, H. Jiang, D. Feng, Z. Niu, K. Zhou, and Y. Wan. DEBAR: A scalable high-performance de-duplication storage system for backup and archiving. In *Parallel & Distributed Processing, 2010 IEEE International Symposium on*, IPDPS '10, pages 1–12. IEEE, 2010.
- [41] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST '08, pages 18:1–18:14, Berkeley, CA, USA, 2008. USENIX Association.