

# Evaluating the Feasibility of Using Memory Content Similarity to Improve System Resilience

Scott Levy and Patrick G. Bridges  
Department of Computer Science  
University of New Mexico  
{slevy | bridges}@cs.unm.edu

Kurt B. Ferreira, Aidan P. Thompson and  
Christian Trott  
Sandia 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 grows, the failure rate increases proportionally. One of the most common sources of failure in large-scale systems is memory errors. 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 feasibility of this approach by examining memory snapshots collected from eight HPC applications. Based on the characteristics of the similarity that we uncover in these applications, we conclude that our proposed approach shows promise for addressing system resilience in large-scale systems.

## Categories and Subject Descriptors

D.4.5 [Operating Systems]: Reliability—*fault-tolerance*

## 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 [29]. 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 [11, 8].

Memory-related errors are one of the most frequently observed sources of node failure in large-scale distributed sys-

\*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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ROSS ’13, June 10, 2013, Eugene, Oregon, USA

Copyright 2013 ACM 978-1-4503-2146-4/13/06 ...\$15.00.

tems [29]. Moreover, power concerns may exacerbate this problem as we consider deploying low voltage memory chips that are more prone to error [6].

In this paper, we present a novel approach for using content similarity in the memory of HPC applications to improve resilience to uncorrectable memory errors. We then evaluate its feasibility by examining the application memory of eight HPC workloads running on a Cray XE6 supercomputer.

## 2. PROPOSED APPROACH

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_bsdiff` [30] 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

ASC Sequoia Marquee Performance Codes [22]	AMG	A parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids [14].
	IRS	Implicit Radiation Solver. Solves the radiation transport equation by the flux-limited diffusion approximation using an implicit matrix solution [20].
DOE Production Applications	CTH	A multi-material, large deformation, strong shock wave, solid mechanics code [25]
	LAMMPS	Large-scale Atomic/Molecular Massively Parallel Simulator. A classical molecular dynamics simulator [27].
Mantevo Mini-Applications [26], [15]	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 [21].
	Sweep3D	Solves a 1-group time-independent discrete ordinates (Sn) 3D cartesian (XYZ) geometry neutron transport problem. [24]

**Table 1: A brief summary of HPC applications used**

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 an 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,<sup>1</sup> 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 [19]. In other operating systems (e.g., the Kitten lightweight kernel [28], older versions of Linux), raising an MCE simply crashes the node.

For each duplicate or similar page, we maintain a description of its reference page(s) (i.e., the other 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_bsdiff` are not symmetric, *every* similar page requires its own patch data.

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

<sup>1</sup>This might happen if, for example, the MCE was raised by a memory scrubber. However, given the analysis in [17] it is not clear that this is a common scenario.

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.

## 3. EVALUATION

We examined memory content similarity in systems running the eight HPC workloads described in Table 1 to determine the viability of our proposed approach. This set of applications are 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 that was assembled expressly for ensuring that key workloads would perform well on the Sequoia supercomputer at Lawrence Livermore National Laboratory. Additionally, our set includes two important U.S. Department of Energy (DOE) production applications: CTH and LAMMPS.

We generated the data presented in this paper by running each application using MPICH on 8 nodes of a Cray XE6 supercomputer. We used 8 processes on each node for a total of 64 MPI ranks.

### 3.1 Data Collection

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 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 snapshot 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` is able to write 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 periodically snapshot memory 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 the majority of pages that corresponding to memory-mapped files in the applications that we considered 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. However, we can, in practice, use pages backed by stable storage as reference pages for other pages in application memory. But because of this asymmetry, we excluded pages that correspond to memory-mapped files to simplify our analysis.

## 3.2 Data Analysis

After we collected snapshots of the applications’ memory, we analyzed them offline. For each snapshot, we walked through the application’s 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. Identifying zero pages is straightforward. However, identifying duplicate and similar pages require more care.

### 3.2.1 Duplicate Pages

Naively, identifying duplicate pages is a  $O(n^2)$  operation. To reduce the cost of identifying duplicate pages, 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 the memory contents of the applications we consider are not adversarial. As a result, by using the birthday problem [16], it can be shown that the likelihood of such an event is exceedingly small (i.e.,  $\approx 10^{-14}$ ) even for very large memory snapshots [33].

### 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 [13]. 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.

As in [13], this is a statistical, heuristic approach. Although there may be more effective ways of identifying similar pages, the fraction of similar pages we identify using this approach is a lower bound on the total number of similar pages in application memory.

## 3.3 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 begin to understand the variation introduced by this approach, we ran our analysis scripts ten times (randomly choosing the four signature locations each time) on the memory snapshots collected for LAMMPS. We observed that 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 begin to understand the extent of the variability in our data collection mechanism, we collected memory snapshots for ten separate runs of LAMMPS. We then used our analysis scripts to categorize 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 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.

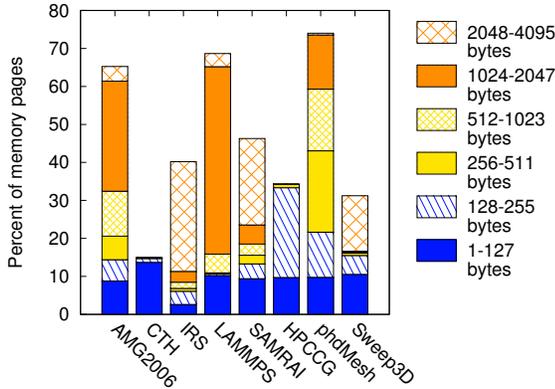


Figure 1: The percent of similar pages for rank 0 as a function of patch size threshold

## 4. RESULTS

In this section, we present the results of our examination. We also discuss why these results are promising for the approach we propose.

### 4.1 Overview

We begin with Figure 2 in which we present 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.

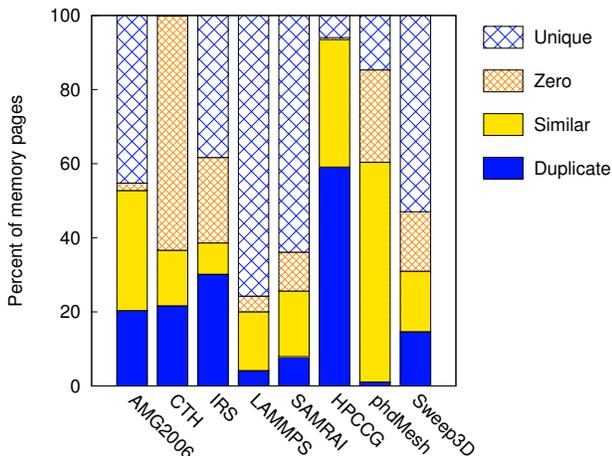


Figure 2: Page categorization within Rank 0 for each application. Each bar represents the page categorization for the memory snapshot that contained the smallest fraction of similar and duplicate pages.

The first observation we make is that five of the applications (AMG, IRS, CTH, HPCCG and phdMesh) exhibit a significant fraction (greater than 35%) of similar and duplicate pages. We also observe that in no case is the fraction of similar and duplicate pages less than 20% of application

memory. More details about page categorization for each application are available in [23].

### 4.2 NUMA

We ran all of our tests 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 [31]. 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 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.

Therefore, for many applications, there may be little incentive to consider application memory within a NUMA domain collectively. However, 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.

### 4.3 Modification Behavior

The approach we propose will also impose a temporal overhead: the time required to maintain metadata will be deducted from the time that the application would otherwise run. The magnitude of the 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 get a sense of how frequently similar and duplicate pages change, we compared the memory contents across the

Application	Rank 0-3			NUMA Domain			$\Delta$ Similar	$\Delta$ Duplicate	% Increase 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.

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.<sup>2</sup> 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 [10]. 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 [23] 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.

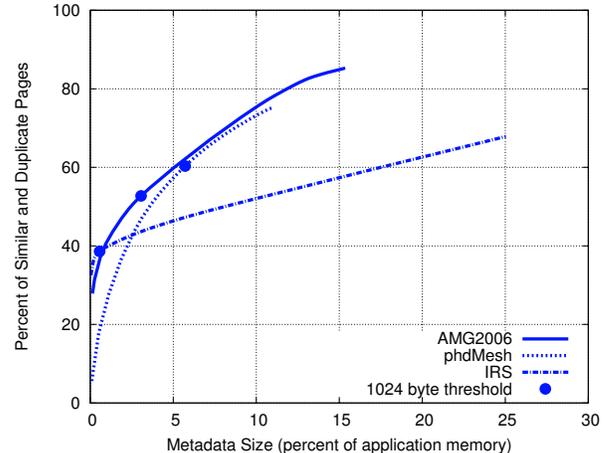
Although many of these results are promising, the results for SAMRAI indicate that there are applications that frequently modify similar and duplicate pages. Unlike the other applications that we considered, a majority of the similar and duplicate pages in the memory of SAMRAI are modified at least once; 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.

<sup>2</sup>This approach may underrepresent the frequency of page modifications because it does not account for modifications that occur between memory snapshots.

#### 4.4 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 3: The percent of similar and duplicate pages as a function of metadata size.** On each of the curves, a solid circle indicates the point on the curve that corresponds to a patch threshold of 1024 bytes.

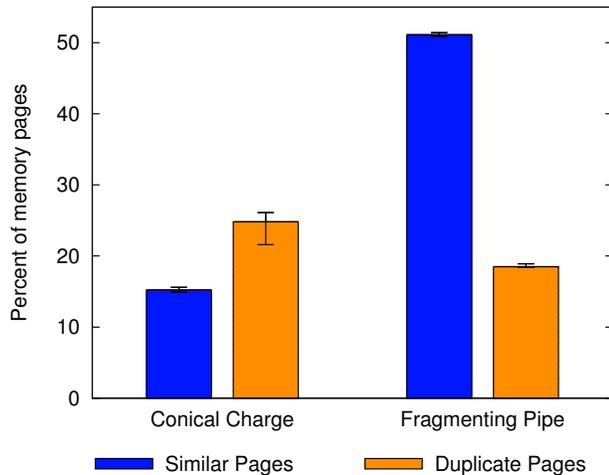
Figure 3 shows the fraction of similar and duplicate pages as a function of metadata size for three applications. The slope of the curves represents the ratio of cost to benefit. For applications like AMG and phdMesh, we can extract significant similarity with a small increase in metadata. In particular, 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 represents an application in which the benefits come at a higher cost.

## 4.5 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 begin to understand the effect of choice of input, 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 shape charge input.



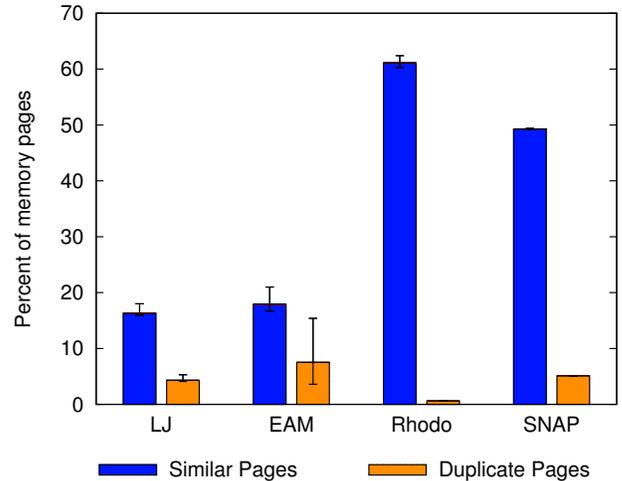
**Figure 4: The effect of different inputs on the fraction of similar and duplicate pages observed in the application memory of CTH**

Figure 4 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 substantially 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) SNAP<sup>3</sup>. All preceding results for LAMMPS presented in this paper were obtained using the LJ potential.

Figure 5 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

<sup>3</sup>SNAP is a computationally intensive potential that uses the same kernel as the GAP potential. [2]



**Figure 5: The effect of using different potentials as input on the fraction of similar and duplicate pages observed in the application memory of LAMMPS**

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.

## 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. In [5], the authors introduced the concept of transparent memory sharing in VMMs. By intercepting disk requests that DMA data into memory, the Disco VMM could consolidate 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, [32] described the broader approach to memory de-duplication that is used in the VMware ESX server. Instead of intercepting disk requests, the authors propose identifying 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 (COW). 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%.

The authors of [33] advocate broadening the scope of sharing in virtualization to consider intranode sharing. To evaluate the feasibility of this approach, they consider the prevalence of intranode sharing between nodes running several HPC applications. For some workloads (notably HPCCG), they observe that significant inter- and intra-node sharing opportunities exist. Based on these promising results, they propose a Content-Sharing Detection System for exploiting intranode sharing in virtualized environments. Similarly,

*SBLLmalloc* has been used to demonstrate that memory consumption can be significantly reduced by consolidating duplicate pages in the application memory of several HPC applications [3].

Most memory de-duplication research has considered consolidating only duplicate pages. However, the Difference Engine [13] introduced the idea 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 [35, 34]. 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 [8, 9] is the most common approach. Asynchronous checkpointing [18, 12] and replication [11] have also been considered. In addition to these system-level approaches, algorithm-based techniques for enabling applications to withstand memory errors have been explored [4, 7]. In contrast, our approach will allow the system to transparently recover from memory errors without requiring application restart or detailed application knowledge.

## 6. CONCLUSION & 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 four 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.

While these results are promising, we have not yet collected data on the impact of this approach on execution runtime. However, based on existing work in memory de-duplication [13, 3] we are optimistic that the execution time overhead will be reasonable. For example, in [13] the authors showed that application performance in systems using the Difference Engine, which also exploits page similarity at runtime, was within 7% of native.

Taken as a whole, these initial results suggest that using memory content similarity may 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.

## 7. 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] 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.
- [4] P. Bridges, M. Hoemmen, K. B. Ferreira, M. Heroux, P. Soltero, and R. Brightwell. Cooperative application/os DRAM fault recovery. *Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids in conjunction with the Euro-Par Conference, Lecture Notes in Computer Science*, pages –, 2011.
- [5] 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.
- [6] 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. DFTVS'08. IEEE International Symposium on*, pages 114–122. IEEE, 2008.
- [7] Z. Chen and J. Dongarra. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, April 2006.
- [8] 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.
- [9] 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.

- [10] 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.
- [11] 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.
- [12] 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*, May 2011.
- [13] 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.
- [14] V. Henson and U. Yang. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41(1):155–177, 2002.
- [15] 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.
- [16] L. Holst. The general birthday problem. *Random Structures and Algorithms*, 6(2-3):201–208, 1995.
- [17] 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.
- [18] D. B. Johnson and W. Zwaenepoel. Recovery in distributed systems using asynchronous and checkpointing. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 171–181, 1988.
- [19] A. Kleen. mcelog: memory error handling in user space. In *Proceedings of Linux Kongress 2010*, Nuremberg, Germany, September 2010.
- [20] 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](https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/irs/irs.readme.html).
- [21] Lawrence Livermore National Laboratories. SAMRAI. <https://computation.llnl.gov/casc/SAMRAI/index.html>.
- [22] Lawrence Livermore National Laboratories. ASC Sequoia Benchmark Codes. <https://asc.llnl.gov/sequoia/benchmarks>, August 2009.
- [23] S. Levy, K. B. Ferreira, P. G. Bridges, A. P. Thompson, and C. Trott. An Examination of Content Similarity within the Memory of HPC Applications. Technical Report SAND2013-0055, Sandia National Laboratories, 2013.
- [24] Los Alamos National Laboratories. Sweep3d. [http://www.c3.lanl.gov/pal/software/sweep3d/sweep3d\\_readme.html](http://www.c3.lanl.gov/pal/software/sweep3d/sweep3d_readme.html), 1999.
- [25] 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.
- [26] Sandia National Laboratories. Mantevo. <http://software.sandia.gov/mantevo>.
- [27] Sandia National Laboratories. The LAMMPS molecular dynamics simulator. <http://lammps.sandia.gov>, April 2010.
- [28] Sandia National Laboratories. Kitten lightweight kernel. <https://software.sandia.gov/trac/kitten>, March 10 2012.
- [29] 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 (DSN2006)*, June 2006.
- [30] A. Tuininga. cx\_bsdiff. [http://starship.python.net/crew/atuining/cx\\_bsdiff/index.html](http://starship.python.net/crew/atuining/cx_bsdiff/index.html), February 2006.
- [31] 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.
- [32] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, Dec. 2002.
- [33] 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.
- [34] 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 (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.
- [35] 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.