SAND2016-0499C

Similarity Engine: Using Content Similarity to Improve Memory Resilience

Scott Levy
Department of Computer Science
University of New Mexico
slevy@cs.unm.edu

Kurt B. Ferreira
Center for Computing Research
Sandia National Laboratories
kbferre@sandia.gov

Patrick G. Bridges
Department of Computer Science
University of New Mexico
bridges@cs.unm.edu

ABSTRACT

In next-generation extreme-scale systems, application performance will be limited by memory performance characteristics. The first exascale system is projected to contain many petabytes of memory. In addition to the sheer volume of the memory required, device trends, such as shrinking feature sizes and reduced supply voltages, have the potential to increase the frequency of memory errors. As a result, resilience to memory errors is a key challenge. In this paper, we introduce the *Similarity Engine*, a lightweight software service for identifying similarity in the memory of many important applications. We evaluate the performance of the Similarity Engine and demonstrate that it is able to identify significant similarity in the memory of HPC applications. Further, we show that by leveraging memory content similarity to improve application resilience, the Similarity Engine can improve application performance by up to 70% on next-generation systems.

CCS Concepts

•**Software and its engineering** → **Checkpoint / restart;**
•**Computer systems organization** → **Reliability;**

Keywords

high-performance computing, resilience, fault tolerance

1. INTRODUCTION

Effective use of memory is a key challenge in next-generation extreme-scale systems [10]. These systems will likely contain many petabytes of memory and will present difficult challenges to system performance, power management, and resilience. In particular, the sheer volume of memory required to build these systems coupled with device trends, such as shrinking feature sizes and reduced supply voltages, have the potential to increase the frequency of memory errors. As a result, current projections suggest that a memory error could happen as frequently as once per hour [20].

HPDC'16 Kyoto, Japan
© 2016 ACM. ISBN 123-4567-24-567/08/06...$15.00
DOI: 10.475/123_4

A number of techniques that exploit content similarity in application memory have been used to address memory challenges in large-scale commodity systems [5, 13]. Despite significant evidence of memory similarity in HPC systems [19], such techniques have been only occasionally studied in HPC systems [3, 29].

In this paper, we present the design, implementation, and evaluation of an HPC-oriented memory similarity service: *Similarity Engine*. This system provides a general, lightweight, application-independent service for detecting and tracking per-node memory similarity in HPC applications. This service enables the systematic implementation of a wide range of memory optimizations, including more general implementations of previous techniques, and new optimizations specifically focused on the challenges presented by the design of exascale systems. As part of this work, we describe the design of one such service for improving resilience to memory errors and evaluate its potential effectiveness based on the Similarity Engine's ability to find and track similarity in a broad range of HPC applications.

In the remainder of this paper, we describe our contributions towards leveraging HPC application memory similarity to dramatically improve system resilience. These contributions include:

- A lightweight, application-independent, HPC-oriented memory similarity tracking system, Similarity Engine, that efficiently *discovers similarity* in the memory of HPC applications. We present an empirical study of key design trade-offs in the implementation of this service.
- A novel, lightweight algorithm (xor+lz4) for evaluating the similarity of memory pages;
- A study that *evaluates similarity* in a diverse set of HPC applications using the Similarity Engine, showing behaviors including: applications with abundant, easy-to-track similarity, applications with significant similarity that is infeasible to track, and applications with minimal inherent similarity;
- A description and evaluation of a technique that *exploits similarity* identified by Similarity Engine to improve the performance and increase the MTTI of HPC applications by repairing uncorrectable memory errors.

Following an explanation of these contributions, we further discuss our results and their implications, describe related work and directions for future work, and conclude.

Application	Description
HPCCG	One of the Mantevo mini-applications [26]. Designed to mimic finite element generation, assembly and solution for an unstructured grid problem.
LAMMPS	Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS). A classical molecular dynamics simulator from Sandia National Laboratories [25]. The data presented in this paper are from experiments that use the Lennard-Jones (LAMMPS-lj) and Embedded Atom Model (LAMMPS-eam) potentials that are included with the LAMMPS distribution.
CTH	A multi-material, large deformation, strong shock wave, solid mechanics code [21] developed at Sandia National Laboratories. The data presented in this paper are from experiments that use an input that describes the simulation of the detonation of a conical explosive charge (CTH-st).
LULESH	Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). A proxy application from the Department of Energy Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx). LULESH approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh [16].
SAMRAI	Structured Adaptive Mesh Refinement Application Infrastructure (SAMRAI). A framework from the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory that is designed to enable the application of structured adaptive mesh refinement to large-scale multi-physics problems

Table 1: Descriptions of the set of workloads that we used for evaluating the performance characteristics of the Similarity Engine.

2. BACKGROUND

We initially presented our idea for exploiting memory content similarity in HPC applications in earlier publications. [17, 19]. Our earlier work used an offline approach to characterize the contents of memory snapshots from several HPC applications [17,19]. These results demonstrated that it was likely that we could efficiently exploit similarity in application memory.

In this paper, we leverage these preliminary results and introduce the implementation of a software library that allows us to efficiently identify and track similarity during the execution of several important HPC applications. We provide a detailed examination of the tradeoffs involved in identifying and tracking similarity in real time as an application executes. We also present and evaluate a technique for exploiting similarity to improve system resilience characteristics.

3. IMPLEMENTING THE SIMILARITY ENGINE

The goal of the Similarity Engine is to discover and exploit process-level memory similarity in HPC applications that can be used to improve their resilience and performance. This section describes the mechanisms that the Similarity Engine uses to identify and track similarity in application memory.

3.1 Overview

The Similarity Engine categorizes all of the pages in an application's memory into four categories–zero, duplicate, similar, and unique–defined as follows:

- *Zero pages:* pages whose contents are entirely zero
- *Duplicate pages:* pages that (a) are not zero pages; and (b) exactly match the contents of one or more other pages
- *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 is smaller than a tunable threshold: the *difference threshold.*
- *Unique pages:* pages that do not fall into any of the preceding three categories

Although they are categorized separately here, when we discuss *similarity* in general we mean the set of pages that are duplicate, zero, or similar.

Based on these definitions, the Similarity Engine works by first tracking application memory allocation and periodically scanning allocated memory. During this scan, it identifies zero and duplicate pages by computing hashes of page contents, and uses efficient heuristics to detect similarity.

3.2 Tracking Application Memory

The Similarity Engine tracks an application's memory allocation by using features of the GNU linker to interpose code between the application and the standard C/C++ memory allocation functions (e.g., `malloc`, `free`, `new`, `delete`). The Similarity Engine maintains a database of the application's memory allocations. Each time the application allocates (or deallocates) memory, the Similarity Engine updates its database to ensure that it has an up-to-date view of the memory that is currently allocated by the application.

3.3 Managing Memory Modification

Exploiting memory contents requires the Similarity Engine to have an accurate picture of which pages have been modified since they were last categorized. To this end, the Similarity Engine uses `mprotect` to make every page of the application's allocated memory read-only. When the application writes to a page of read-only memory, a segmentation fault occurs. The Similarity Engine uses a `SIGSEGV` signal handler to receive notification of each segmentation fault. The Similarity Engine's signal handler examines the fault that occurred to identify the page of memory that is being

System	Nodes	Sockets/Node	Processors	Operating System	Interconnect	MPI
Compton	42	2	Intel Sandy Bridge (2.6 GHz / 8 cores)	Linux 2.6.32	Mellanox MT26428 QDR InfiniBand HCA	OpenMPI 1.8.1
Chama	1,232	2	Intel Sandy Bridge (2.6 GHz / 8 cores)	Linux 2.6.32	QLogic QLE7340 QDR InfiniBand HCA	OpenMPI 1.8.4

Table 2: Configuration details of clusters used to gather experimental data.

written. It then updates its metadata accordingly, restores write privileges to the accessed page, and returns control to the application.

After write privileges have been restored to a memory page, the Similarity Engine no longer knows its relationship to other pages of memory. As the application executes and more and more pages are written to, the Similarity Engine's knowledge of the state of the application's memory diminishes. To re-establish its knowledge of inter-page relationships, the Similarity Engine periodically restores the application's memory pages to read-only. It accomplishes this by dividing the application's execution time into tunable *protection intervals*. For the data presented in this paper, the protection interval is 60 seconds long. The Similarity Engine is notified by a `SIGALRM` signal at the end of each interval. At the beginning of each protection interval, the Similarity Engine removes write privileges for the memory allocated by the application, identifies the set of pages that are similar, and computes the MD5 hash of every page that has been accessed since the beginning of the preceding interval. Because we assume that the contents of application memory are not adversarial, two pages with same hash value will have the same contents with very high probability. Similarly, if the hash value of a page has not changed, then it is highly likley that its contents have not been modified.

The Similarity Engine can also be configured to periodically examine and collect statstics about the state of the application's allocated memory. Each protection interval can be subdivided into tunable *sample intervals*. At the end of each sample interval, the Similarity Engine examines the application's memory to identify the current state of its similarity relationships. For all of the statistical data presented in this paper, the sample interval is 20 seconds.

3.4 Handling System Calls

When read-only memory is passed to the kernel in a system call, writing to this memory does not invoke the Similarity Engine's user-level segmentation fault handler. As a result, system calls may fail. If return values are not carefully checked by the application, the consequences of the failure may be difficult to predict.

To determine the extent to which the applications examined in this paper may attempt to pass references to read-only memory to the kernel, we created a version of the Similarity Engine that leverages Linux's `ptrace` mechanism to intercept and inspect the system calls made by the application. Specifically, this version of the Similarity Engine creates a child thread during its initialization and uses `ptrace` to attach to the application (its parent). Each time the application enters a system call, the child examines the contents of the registers that contain the arguments being passed to determine whether they contain a reference to memory that the Similarity Engine is actively tracking. For the set of applications and associated input decks considered in this paper, no references to tracked memory were found in the arguments passed to any system call.

There are likely applications for which references to user-allocated heap memory are passed to system calls. In these cases, efficiently identifying similarity without jeopardizing the correctness of the the simulation will likely require integrating the Similarity Engine into the kernel.

3.5 Passing memory references to MPI

Passing read-only message buffers to the MPI library can result in unpredictable behavior. The Similarity Engine uses the PMPI profiling layer to intercept MPI calls that reference one or more message buffers. For each message buffer, the Similarity Engine determines whether the buffer occupies memory that it is tracking (i.e., whether it is comprised of memory that is or may become read-only). If so, the Similarity Engine makes it writable and updates its metadata to reflect the fact that it is no longer tracking the state of this memory. As a result, the pages that comprise the message buffers passed to MPI are categorized as unique in all of the statistics presented in this paper. The Similarity Engine currently lacks a mechanism for resuming the tracking of memory used for MPI message buffers.

4. DISCOVERING SIMILARITY

The performance of the Similarity Engine depends on how efficiently it is able to discover pairs of similar pages. This section evaluates the tradeoffs involved in computing differences between pages and identifying pairs of potentially similar pages.

4.1 Experimental Setup

This paper presents results from experiments conducted to evaluate the potential costs and benefits of exploiting memory content similarity. The characteristics of six workloads are considered. These workloads, described in Table 1, include two important DOE production applications (LAMMPS and CTH), a proxy application (LULESH) from the Department of Energy's Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx), a mini-application from Sandia's Mantevo suite (HPCCG) and an example application from an important library used in large-scale DOE production applications (SAMRAI). For LAMMPS, we consider two input decks. We use 4 KiB memory pages throughout.[1]

For all of the applications except for CTH, the experiments were performed using Compton, a Linux Infiniband

[1]Throughout this paper, we use the binary prefixes defined by the International Electrotechnical Commission (IEC) to indicate binary orders of magnitude. For example, a KiB is a *kibibyte*, 2^{10} bytes.

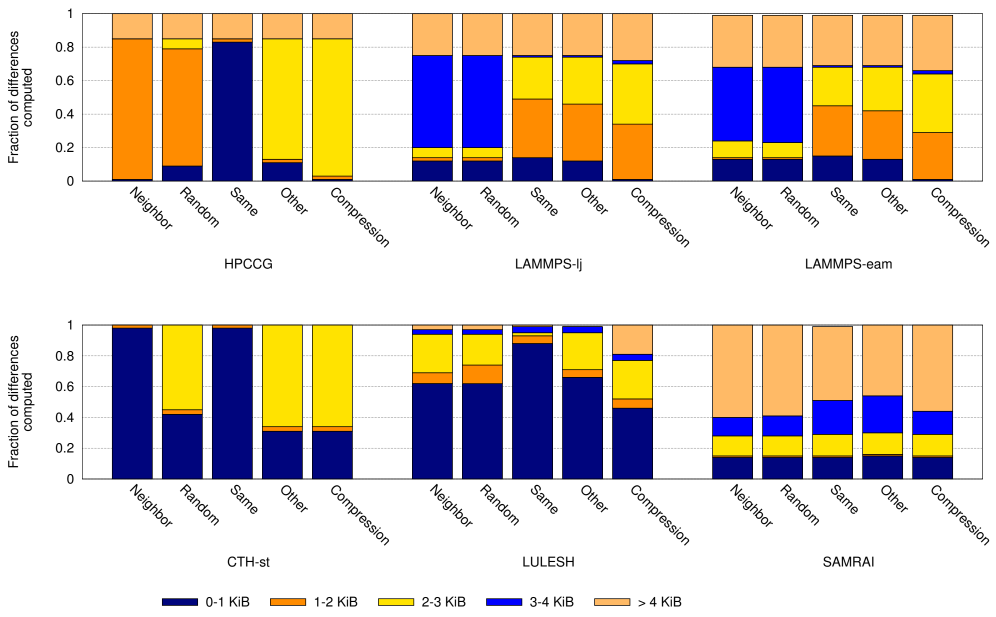


Figure 1: Similarity heuristic microbenchmark. For each target application a sequence of snapshots of allocated memory were collected. Up to 5,000 candidate pages are randomly selected from each snapshot of each application. For each candidate, differences are computed between it and all of the other pages in memory (differences between pairs of pages that are identical are not computed). The resulting set of differences is binned based on their size. This figure shows the size distribution for each of five categories of differences. *Neighbor* is the set of differences that are computed between two pages that are adjacent in the virtual address space and belong to the same allocation. *Random* is the set of differences the candidate page and each of two pages chosen randomly from the same memory allocation as the candidate page. *Same* is all of the pages (except for the neighbors) in the same memory allocation as the candidate page. *Other* are the pages that are not in the same memory allocation as the candidate page. *Compression* is the result of compressing each candidate page using lz4.

cluster. Because CTH is export-controlled, the CTH experiments were performed on Chama, also a Linux Infiniband cluster. Details on the composition and configuration of these clusters are presented in Table 2.

The remainder of this section evaluates the effectiveness and costs of computing differences and identifying potentially similar pages. It does so by presenting and examining the results of a series of small-scale experiments. These experiments were conducted by running each of the seven workloads on 8 MPI processes across 4 nodes of the clusters described above.

4.2 Computing Page Differences

At the beginning of each protection interval, the Similarity Engine identifies similarity by computing differences between pairs of pages. This section evaluates the performance characteristics of five algorithms for computing page differences.

These algorithms include two well-known delta encoding algorithms, a novel lightweight differencing algorithm, and two page compression algorithms.[2] The differencing algorithms allow the Similarity Engine to exploit redundancies that exist between pairs of pages. Page compression enables the Similarity Engine to exploit redundancies within a single page of memory. A description of each of the five differencing algorithms follows:

- *bsdiff* (and its mirror, *bspatch*) uses suffix sorting and bzip2 to compute differences between pairs of binary files [7].
- *Xdelta* is an open-source delta encoder/decoder based on VCDIFF [14]. VCDIFF is both an algorithm and a format for encoding the differences between binary files.
- *lz4* is a lightweight, lossless data compression algorithm that is based on LZ77 compression [2].
- *xor+lz4* is a novel lightweight differencing algorithm that combines naive differencing (bit-wise exclusive-or) with lightweight compression (lz4).
- *bzip2* is a lossless data compression algorithm that uses Burrows-Wheeler transforms and Huffman coding [1].

These algorithms compute the difference between a *candidate page* and a *reference page*. This difference would allow the Similarity Engine to recreate the contents of the candidate page from the reference page. Because xor+lz4 generates symmetric differences, the differences it generates to can be used to reconstruct either the reference page or the candidate page from the other. The other four algorithms

[2]In the case of page compression, the compressed page can be viewed as the difference between a candidate page and a null page.

generate asymmetric differences (i.e., the difference between the candidate page and a reference page can only be used to reconstruct the candidate page). As a result, xor+lz4 requires the computation of half as many differences as bsdiff and Xdelta. Because the two compression algorithms, bzip2 and lz4, consider the difference between an application page and a null page, the Similarity Engine never has to compute more than one difference per page of application memory.

Efficiently identifying similarity in application memory requires that the Similarity Engine be able to quickly encode small differences between pairs of pages. The speed of difference encoding strongly influences the runtime overhead of the Similarity Engine. The size of the differences dictates its memory overhead. To limit the memory overhead, similarity is defined relative to a tunable *difference threshold*. A memory page is similar only if the difference between it and a reference page falls below this threshold. The difference threshold represents a tradeoff between identifying more similarity and memory overhead. Increasing the difference threshold means that more pages will be similar, but also that more memory must be devoted to storing differences.

Algorithm	Mean encode time (μs)	Mean decode time (μs)	Efficacy (compressed bytes/μs)
XOR+lz4	19.57	3.58	70.41
lz4	14.12	2.39	155.67
xdelta	182.92	15.15	12.69
bsdiff	3474.69	612.25	0.73
bzip2	1381.12	174.56	1.92

Table 3: Temporal costs of identifying similarity. This table shows the average time required to encode and decode differences. To examine the relationship between difference size and difference encoding speed, it also examines the *efficacy* of each algorithm in terms of the number of bytes it is able to compress out of the corpus of candidate pages per microsecond.

The suitability of these algorithms for identifying similarity was examined with two microbenchmarks. To run these microbenchmarks, we constructed a library that takes periodic snapshots of the allocated memory of each of our target workloads. We then randomly chose a maximum of 5,000 candidate pages—that were neither zero nor duplicate—from each snapshot of each application. For each candidate, we computed the difference between it and every other page in the same snapshot.

The first microbenchmark measures the speed of difference encoding and decoding. The results are shown in Table 3. The fastest algorithms by a substantial margin are xor+lz4 and lz4. They are more than eight times faster than Xdelta and nearly 200 times faster than bsdiff. Moreover, because Xdelta, bzip2, and bsdiff generate asymmetric differences, they would require the Similarity Engine to compute twice as many differences as xor+lz4 for the same number of pages.

The second microbenchmark measures the size of the differences generated by each algorithm. For each difference algorithm, the smallest difference computed for each candidate was recorded. The results are shown in Figure 2. In this figure, a point at (x, y) indicates that for a y fraction

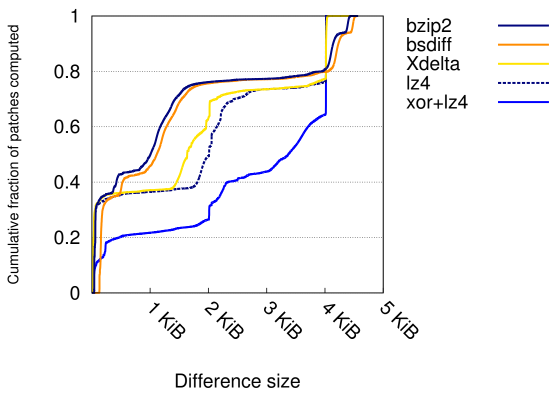


Figure 2: Difference size microbenchmark. The distribution of the size of the differences computed by the four difference algorithms considered. These data represent the result of computing the difference between 5,000 random pairs of pages from each snapshot of each application.

of the candidates were considered, the size of the smallest difference was less than or equal to x bytes.[3] As this figure demonstrates, the speed of xor+lz4 comes with a cost. It generates substantially fewer small differences than the other three algorithms. As a result, for a fixed difference threshold, xor+lz4 will tend to identify fewer similar pages. The differences generated by lz4 tend to be smaller than those generated by xor+lz4. This due in part to the fact that this microbenchmark considers random pairs of pages. The likelihood of randomly choosing a pair of pages such that they contain redundant information is low. This also demonstrates the importance of developing effective techniques for identifying pairs of pages that are likely to be similar.

These results demonstrate that there is a tradeoff between the speed of computing a difference and the average size of the difference that is generated. The fastest algorithms tend to generate the largest differences. To quantify this tradeoff, Table 3 shows the *efficicay* of each of the give differencing algorithms: the mean number of bytes per microsecond that each is able to compress out of the corpus of the candidate pages. These results show that although xor+lz4 and lz4 generate larger differences, they are able to identify similarity much more efficiently than the other three algorithms.

Computing the difference between two pages with bzip2 or bsdiff requires more than a millisecond. For applications with even modest memory footprints, neither of these approaches are viable due to their speed. Because the xor+lz4 and lz4 algorithms are significantly faster than even Xdelta, we selected them for the Similarity Engine. Although this choice means that the Similarity Engine will find fewer similar pages, it is not clear that the benefits of identifying more similarity with one of the other three algorithms would outweigh the costs.

[3]In principle, the difference between any pair of 4 KiB pages can be captured in 4 KiB (i.e., as the bitwise exclusive-or of the two). However, as this figures shows, none of these algorithms use this optimization. As a result, some differences produced by these algorithms are larger than 4 KiB.

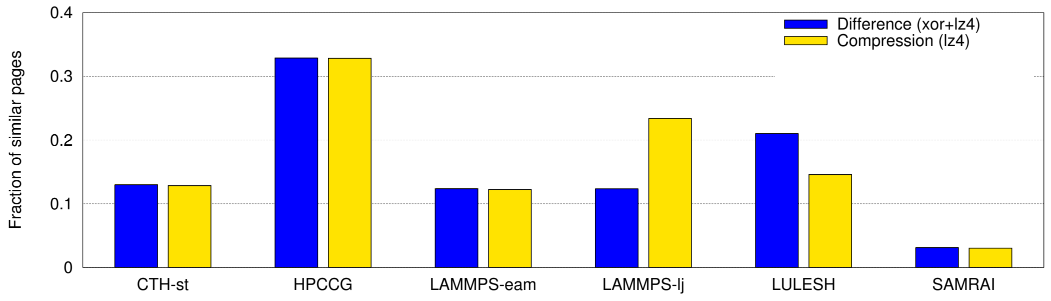


Figure 3: Prevalence of Intra- and Inter-page Similarity. A comparison of the mean fraction of similar pages that the Similarity Engine is able to identify by exploiting intra-page similarity (compression) and inter-page similarity (differencing). These data use a difference threshold of 3 KiB.

4.3 Finding Potentially Similar Pages

Due to the cost of exhaustively computing differences between the pages in an applications memory, an efficient method for identifying pairs of pages that are likely to be similar (i.e., pairs of pages for which the difference between them is small) is necessary. We examined four heuristics for identifying potentially similar pages:

- *Neighbor*: the pages within the same memory allocation that are immediately adjacent to the candidate page in the application's virtual address space;
- *Random*: two pages chosen randomly from the same memory allocation as the candidate page;
- *Same*: all of the pages (except for the neighbors) in the same memory allocation as the candidate page; and
- *Other*: all of the pages that are not in the same memory allocation as the candidate page.

The costs of these four heuristics vary widely. The Neighbor and Random heuristics each identify two potentially similar pages for each candidate, requiring the computation of two differences. The Same and Other heuristics may each identify thousands or tens of thousands of potentially similar pages for each candidate, requiring the computation of potentially thousands of differences. We used a microbenchmark to determine the relative effectiveness of these approaches. Using the same set of snapshots described above, we randomly chose up to 5,000 candidate pages—that are neither duplicate nor zero—from each snapshot. For each candidate, we use xor+lz4 to compute the difference between it and the pages in the sets of potentially similar pages identified by each of the four heuristics described above. Within each of these categories, the size of the smallest observed difference is recorded. The results are presented in Figure 1. In addition to the four heuristics that we have described, we also consider the distribution of the difference sizes when we use page compression. This figure shows that there are limits to the benefits of considering more potentially similar pages. Considering pages within the same allocation as the candidate page is no worse than considering all of the application's other allocations. For all but CTH-st, this requires the computation of many fewer differences. For the two LAMMPS problems and, to a lesser extent, HPCCG, the Same and Other heuristics identify much smaller differences. But for CTH-st, LULESH, and SAMRAI, the distribution of difference sizes generated by the Neighbor heuristics is very similar to the distribution generated by the much more expensive Same heuristic. This figure also shows that for four of our workloads, the neighbor heuristic enables xor+lz4 to outperform lz4. However, for the two LAMMPS problems, lz4 tends to identify smaller differences.

Based on the results of this microbenchmarks, we implemented the Similarity Engine to use the neighbor heuristic for choosing potentially similar pages. Although this choices reduces the amount of similarity that the Similarity Engine is able to identify, it significantly reduces the cost of discovering similarity. It seems unlikely that the Similarity Engine would be viable if every candidate page required the computation of thousands or tens of thousands of differences.

5. EVALUATING SIMILARITY

This section evaluates the amount of similarity that Similarity Engine can discover and track in HPC applications, and measures the cost of tracking this similarity. These experiments used the set of workloads described in Table 1. We ran each workload, except LULESH, with 128 processes on 16 nodes of these clusters. Because LULESH requires the number of processes be a perfect cube, it ran with 125 process. Each experiment is repeated ten times. We perform this entire sequence of experiments twice: once using xor+lz4 to identify similar pages, and once using lz4 to identify similar pages. The data presented in this section required 1200 experiments to be conducted in which the Similarity Engine was linked against one of the workloads. Approximately, 10% (116/1200) of these experiments failed to complete. We discarded all of the data collected during these failed experiments and repeated the experiments.

5.1 Identifying Similar Pages: Compression versus Differencing

As discussed in Section 4.2, we can identify similar pages in application memory by exploiting similarities within a page using compression or across pages using a differencing algorithm. The microbenchmark data in that section suggest that for some applications, the Similarity Engine may be able to identify more similar pages by using compression (lz4) rather than a differencing algorithm (xor+lz4). In this subsection, we compare the number of similar pages in our full-scale tests that the Similarity Engine is able to iden-

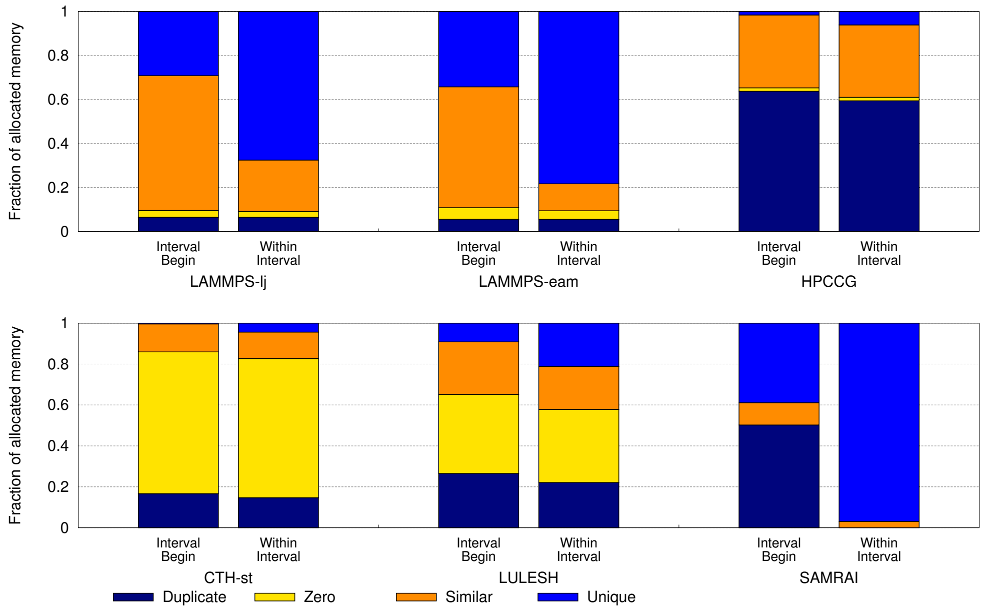


Figure 4: Page categorization. A comparison of the average page categorization at the beginning of a protection interval and the average within a protection interval. When a page is modified, the Similarity Engine no longer knows its relationship to other pages in memory. As a result, it must be categorized as a unique page. These data use xor+lz4 and a difference threshold of 3 KiB.

tify in the memory of our six workloads by using these two techniques to exploit intra- or inter-page similarities. Figure 3 shows the mean fraction of memory pages that the Similarity Engine identifies as being similar based on whether it uses compression (lz4) or differencing (xor+lz4). This figure shows that for four (CTH-st, HPCCG, LAMMPS-eam, and SAMRAI) of the six workloads the difference between the two approaches is negligible. In other words, the volume of redundant information in the memory of these workloads is approximately equal within and across memory pages. However, the LAMMPS-lj and LULESH results do reflect a significant difference based on the method chosen. Compression allows the Similarity Engine to identify nearly twice as many similar pages in LAMMPS-lj as differencing does. In contrast, differencing allows the Similarity Engine to identify nearly 50% more similar pages in the memory of LULESH. Based on these results, the results presented in the remainder of this section use lz4 to identify similar pages in the memory of the two LAMMPS problems and xor+lz4 to identify similarity in the memory of the other four workloads.

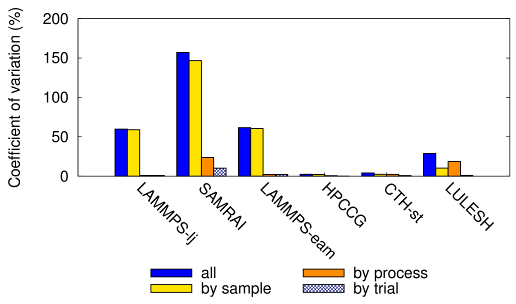
5.2 Prevalence of similarity

To evaluate how effectively our library can identify similarity, we conducted a sequence of tests with each of our applications linked with the Similarity Engine. These experiments generated a three-dimensional dataset. The three dimensions are: *samples*, the Similarity Engine collects statistics periodically (once per sample interval) as the application executes; *processes*, the Similarity Engine collects statistics for each application process; and *trials*, each experiment was repeated ten times. The results of our examination are shown in Figure 4. This figure shows the mean fraction of the pages in application memory that fall into the four categories that we have defined. For each application, we show the mean fraction of memory in each category for samples taken at the beginning of a protection interval. We also show the mean fraction of memory in each category for samples taken within a protection interval. Each time the application writes to a similar, duplicate, or zero page, we no longer know its state or its relationship to other pages in memory. Because accessed pages must be classified as unique, the prevalence of similarity necessarily decreases between protection intervals.

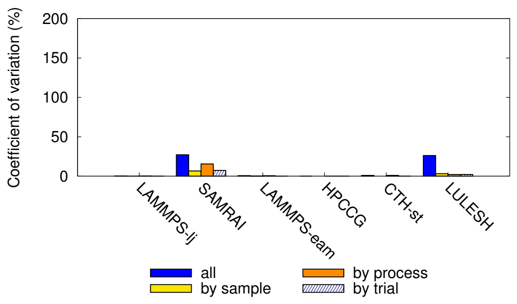
For each application, we find significantly more similarity at the beginning of a protection interval than within the interval. The difference is particularly stark for SAMRAI; on average nearly 60% of memory is similar, duplicate or zero at the beginning of a protection interval, these categories comprise less than 6% of memory within an interval. The difference is less significant for LAMMPS-lj and LAMMPS-eam and they exhibit more similarity within an interval. The differences are more modest for the other three applications. For CTH-st, HPCCG, and LULESH we see significant similarity even within a protection interval.

To measure the variability in the data shown in Figure 4 we computed the coefficient of variation[4] over each dimension of the dataset. The results are shown in Figure 5. Figure 5(b) shows the variation we observe in the total number of pages allocated by each allocation. The total number of memory pages allocation by these workloads is relatively constant across all dimensions. In particular, there is little variability in these data across trials; repeated experiments yield consistent results. Figure 5(a) shows the variation we observe in non-unique pages. The fraction of memory that is comprised of non-unique (similar, duplicate and zero) pages

[4]The *coefficient of variation* is the standard deviation (σ) divided by the mean (μ).

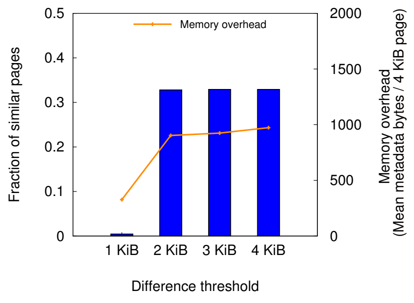


(a) Non-unique pages

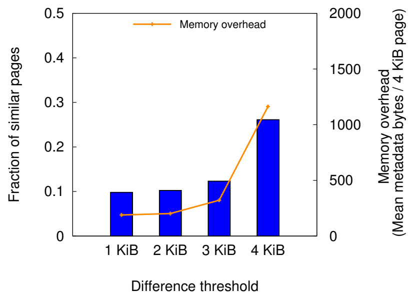


(b) Total pages

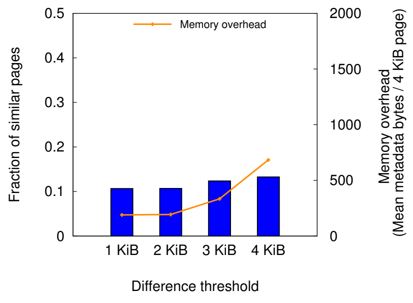
Figure 5: Page category variability. An examination of the variability of the composition of each application's memory using a 3072-byte difference threshold. Non-unique pages are the set of pages that are duplicate, zero, or similar. *All* is the value of the coefficient of variation when computed over the entire dataset. *By process* represents how much variation exists across application processes. *By sample* represents the variation over the lifetime of the application. *By trial* represents the variation across successive executions of the application.



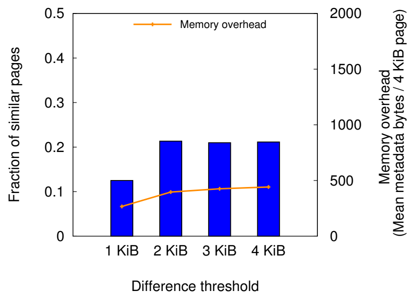
(a) HPCCG



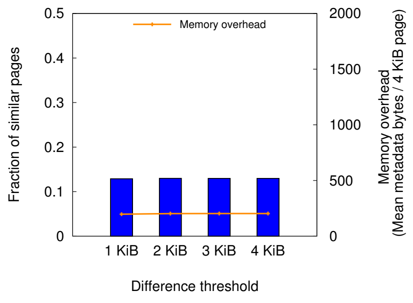
(b) LAMMPS-lj



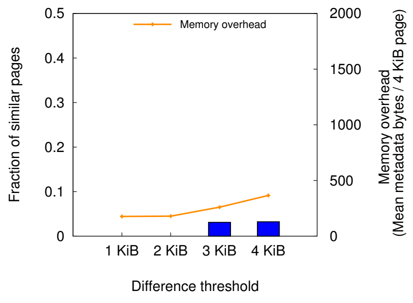
(c) LAMMPS-eam



(d) LULESH



(e) CTH-st



(f) SAMRAI

Figure 6: Difference Threshold Benchmark. For each target application, we consider the trade off between the number of similar pages and memory overhead. As we increase the difference threshold and allow for larger differences, the memory overhead increases.

is a good proxy for the variability of the potential benefits of exploiting memory content similarity. Figure 5(a) demonstrates that the fraction of non-unique pages is also consistent across trials. There is also very little variation in the fraction of non-unique pages in the other dimensions of the dataset for several of the applications. One exception is SAMRAI. SAMRAI exhibits signficant variation in the fraction of non-unique pages that comprise its memory. This behavior is largely due to the fact that we see significant similarity in SAMRAI's memory at the beginning of a protection interval and very little within an interval. Another factor is that a significant fraction of SAMRAI's memory is composed of unique pages. As a result, small changes in the absolute number of duplicate, similar, and zero pages can result in large variations in their respective fractions. The other two exceptions are the two LAMMPS workloads. The variation in these data is due largely to the fact that the Similarity Engine identifies much more similarity at the beginning of the protection interval than within the interval.

5.3 Memory overhead

Effectively exploiting similarity requires that we maintain metadata about the memory currently allocated. As we identify pairs of similar pages, we need to store the encoded difference and the address of the reference page. There is a tradeoff between the difference threshold and the resulting memory overhead. Increasing the difference threshold identifies more similar pages, but results in the retention of more and larger differences.

To examine this tradeoff, we ran each of our six target applications with four difference thresholds: 1 KiB, 2 KiB, 3 KiB, and 4 KiB. For each run we also tracked the memory overhead. The results are shown in Figure 6. We observe that for three of the applications (CTH-st, LULESH, and SAMRAI) the memory overhead is quite stable as we consider larger difference thresholds and small: on average, a few hundred bytes of metadata per page. HPCCG requires significantly more metadata, but this is due in part to the fact that much of its memory is comprised of non-unique pages. For LAMMPS-lj and LAMMPS-eam, increasing the difference threshold above 2 KiB results in a significant increase in the fraction of similar pages and the memory overhead.

5.4 Runtime overhead

Identifying similarity using these techniques requires that we occasionally interrupt the application. We interpose metadata maintenance operations between the application and the standard C/C++ memory allocators. We restore write privileges as accesses to read-only pages generate segmentation faults. As each protection interval begins, we must also change the access privileges for pages of allocated memory to be read-only and identify similar pairs of pages. For each of our target workloads, we measured the inflation of the application's execution time. The results are presented in Table 4. To examine the cost of identifying similar pages, this table includes the runtime overhead of the Similarity Engine when it is configured to identify similar, duplicate, and zero pages (all similar) and the overhead when is configured just to identify duplicate and zero pages (duplicate only). It also shows how frequently each application writes to read-only memory when linked with the Similarity Engine and how frequently each application allocates memory.

For three of the applications, CTH-st, HPCCG, and SAMRAI, the runtime overhead is below 3%. LAMMPS-lj and LAMMPS-eam have the highest runtime overhead. Part of the reason for this overhead is that these two application write to read-only memory much more frequently than any of the other applications. Each time an application writes to read-only memory, it incurs the costs associated with invoking the Similarity Engine's signal handler. Additionally, at the end of the protection interval, the Similarity Engine has to re-evalaute whether the page is similar, duplicate, or zero. LULESH also incurs relatively high overheads. This is due in part to the fact that it allocates memory much more frequently than any other the other applications. Each time the application allocates (or deallocates) memory, the Similarity Engine must update its metadata.

6. EXPLOITING MEMORY SIMILARITY TO CORRECT MEMORY ERRORS

Uncorrectable DRAM errors have been shown to be a significant source of failure on current and future leadership-class HPC systems [28]. When an uncorrectable ECC error is detected on a modern 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, if the fault occurred on a page whose contents are backed up by disk (e.g., a clean page in the page cache), the error can be handled by invalidating the appropriate cache or page table entry. 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 [15]. In other operating systems (e.g., the Kitten lightweight kernel [27], 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 encoded difference. When an uncorrectable memory error occurs on a similar, duplicate, or zero page, we can use the metadata that we have collected to reconstruct the damaged page. Reconstructing a duplicate page is straightforward. We simply restore the contents of the damaged page from the contents of one of its reference pages.[5] For zero pages, we can simply replace the damaged page with a page filled with zeros. For similar pages, the process is only slightly more complex. We can reconstruct the contents of the damaged page by applying the difference stored in our metadata to the associated reference page. As shown in Table 3, using xor+lz4 to reconstruct a damaged page by decoding the difference stored in our metadata is extremely fast: on average $3.58\mu s$ per 4 KiB page. Using lz4 is slightly faster: $2.39\mu s$ per 4 KiB page.[6]

We observe that the additional memory required for our metadata does not significantly increase the vulnerability of the application to memory errors. An error in our metadata does not affect the continued operation application. Moreover, if an error occurs in the set of stored differences (which is the majority of our metadata), we can, with high probability, invalidate the difference and regenerate it at the beginning of the next protection interval. In the worst case, we can take a proactive checkpoint (to eliminate lost work that would need to be re-executed) and restart the application.

Exploiting similarity in this way allows the application to continue execution when an otherwise uncorrectable memory error occurs rather than restarting and rolling back to the last checkpoint. This increases the mean time to interrupt (MTTI) of the system. Using an existing model of rollback avoidance [18][7], we can model how the increased MTTI would affect the performance of an application executing on a hypothetical next-generation system (*see* Table 5). Based on the characteristics of this system, the predicted performance of next-generation applications whose memory characteristics are represented by one of our six target applications is shown in Figure 7. This figure examines the po-

[5]In practice, it may be prudent to reconstruct the page in a different physical location in memory

[6]It is the absolute difference between the two that matters. When an uncorrectable memory error occurs, the Similarity Engine will only need to reconstruct one memory page.

[7]Rollback avoidance is the set of techniques that allow an application to continue to execute, rather than rolling back to a previous checkpoint, when a failure occurs.

Application	Runtime Overhead (all similar)	Runtime Overhead (duplicate only)	Write Exceptions Per Second	Allocations Added Per Second
CTH-st	2.76%	1.99%	485.46	0.06
HPCCG	1.26%	1.30%	54.31	0.06
LAMMPS-eam	14.27%	11.11%	2494.43	1.65
LAMMPS-lj	13.12%	9.95%	1985.54	1.96
LULESH	9.90%	8.89%	31.42	3319.75
SAMRAI	1.84%	1.58%	67.64	594.68

Table 4: Median runtime overheads (ratio of median execution times) using xor+lz4, a 3 KiB difference threshold, and a 60 second protection interval. The coefficient of variation for the execution time data used to generate this table is below 1.5% for CTH-st, HPCCG, LAMMPS-eam, LAMMPS-lj, and SAMRAI. The coefficient of variation for the "all similar" runtime overhead of LULESH is approximately 22%. The coefficients of variation for the other LULESH experiments is less than 1%.

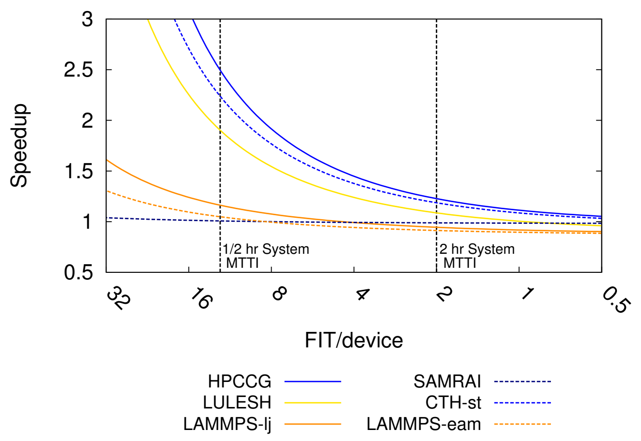


Figure 7: Application speedup. We use the average fraction of memory that is duplicate, zero or similar during a protection interval and an existing model of rollback avoidance [18] to predict application speedup for next-generation applications whose memory characteristics resemble one of our four applications. To generate this figure, we used the runtime overhead from Table 4 and the probability of correcting a memory error from Figure 4. We considered a hypothetical extreme-scale system whose characteristics are shown in Table 5. The x-axis corresponds to the reliability of a single DRAM device in FIT, failures per billion device-hours of operation. The lines in this figure predict the performance of applications whose memory contents look like each of our six workloads.

tential benefit of this approach as a function of memory reliability, a concern for future extreme-scale systems. We measure memory reliability in terms of the failures in time (FIT) per DRAM device: the number of expected failures per billion hours of device operation. We consider a system comprised of nodes whose MTBF is approximately 100 years when memory failures are excluded. A system comprised of these nodes would have an MTBF of approximately 7 hours. We consider a range of potential values for memory reliability, from very reliable (one or two memory failures in the system per day) to very unreliable (several memory failures

PARAMETER	VALUE
nodes	131,072
total system memory	32 PiB
memory devices/node	1152
FIT/node (excluding memory)	1100
checkpoint commit time	10 minutes

Table 5: Characteristics of the hypothetical next-generation, extreme-scale system used to generate Figure 7. A FIT value of 1100 corresponds to an MTBF of approximately 100 years The checkpoint commit time assumes that checkpoints are written to a parallel file system. Its value is based on existing studies of checkpoint performance [9, 22].

in the system per hour). This range of memory reliability is consistent with existing projections of memory performance on future systems [8, 20, 28].

As this figure shows, exploiting similarity to correct memory errors is effective for some but not all memory-use profiles. Applications whose memory-use patterns resemble those of SAMRAI are unlikely to see much benefit from this approach. However, applications that use memory like CTH-st, HPCCG, and LULESH do can potentially see substantial increases in application execution speed. If future memory devices fall on the unreliable end of this range, as some predict for future leadership-class systems, then applications with memory-use patterns that are similar to LAMMPS-lj and LAMMPS-eam may also see significant gains.

7. RELATED WORK

7.1 Memory content similarity

Duplicate memory regions have been successfully exploited to reduce memory consumption in the context of virtualization for decades [5]. The Difference Engine relaxed the requirement of exact duplicates and introduced the notion of similarity [13].

More recently, techniques have been proposed for exploiting similarity in HPC applications. SBLLmalloc reduces the memory usage of HPC applications by sharing duplicate pages between processes [3].[8] ConCORD is a distributed

[8]Many papers use the word "similarity", but in most cases,

service for managing duplicate pages across nodes. As an example of how their service can be used, the authors present an implementation of collective checkpointing [29]. The Similarity Engine, as described in this paper, is the first documented use of memory similarity for something other than de-duplication.

Although we have previously examined the potential for exploiting similarity in HPC applications [19], the Similarity Engine is the first implementation of a system for identifying similar (not just duplicate) pages in application memory. Our earlier work relied on offline analysis of memory snapshots using a powerful, but expensive, differencing algorithm (bsdiff). In this paper, we identified two lightweight algorithms, lz4 and xor+lz4, that enabled us to implement the Similarity Engine, a lightweight software library, that is able to efficiently identify significant similarity in the memory of HPC applications.

7.2 Correcting memory errors

Given the dire predictions about the frequency of memory errors, substantial effort has been devoted to developing mitigation techniques. These techniques include: exploiting algorithmic properties to correct errors [4, 6], predicting the occurrence of failures [11, 12] and replication of instructions [23, 24] or entire processes [9].

Our approach is neither a replacement for nor a direct competitor with these approaches. We present a novel approach for exploiting memory similarity to correct memory errors that could be used in conjunction with any of these existing techniques to further improve application resilience to memory errors.

8. FUTURE WORK

In this paper, we have focused on exploiting memory content similarity to correct memory errors. However, there are other ways to exploit the similiarity identified by the Similarity Engine. For example, we can use similarity to reduce the volume of system-level checkpoints. A checkpoint need only include a single copy of each duplicate page and a pair of similar pages can be replaced by one of the pages and the difference between the two. Similarity can potentially also be used to detect silent data corruption. During an application's execution, the difference between two similar pages can be used to ensure that neither page changes without explicitly being written to.

9. CONCLUSION

We introduced a memory similarity service, Similarity Engine, and demonstrated that it can be used to identify significant similarity in application memory: 94% in HPCCG, 87% in CTH-st, and 79% in LULESH. For other applications, the similarity is more modest: 32% in LAMMPS-lj, and 21% in LAMMPS-eam. We also observed that there are applications for which we are unable to identify significant similarity using our current approach (e.g., SAMRAI).

In this paper, we showed how similarity can be used to improve performance by increasing application resilience to memory errors. Specifically, we showed that for extreme-scale systems where memory failures are projected to occur frequently we can exploit similarity to increase application performance by more nearly 70% for HPCCG, 55% for CTH-st and 36% for LULESH.[9]

We showed that the Similarity Engine imposes very low overhead due to very efficient methods for identifying pairs of potentially similar pages and computing differences. We introduced a novel method (xor+lz4) for computing differences that is substantially faster than existing differencing algorithms and allows the Similarity Engine to find nearly 50% more similar pages in the memory of LULESH than compression alone. We also presented a lightweight heuristic (neighbors) that allows the Similarity Engine to cheaply and effectively identify pairs of potentially similar pages in the memory of HPC applications.

The benefits of our approach could be significantly improved with the development of even more efficient differencing algorithms. The algorithms we used for our full-scale experiments, xor+lz4 and lz4, are very fast but the tradeoff is that they identify much less similarity than the other algorithms we considered. Similarly, although our simple neighbor heuristic is surprisingly effective, a more sophisticated heuristic for identifying pairs of potentially similar pages would also improve the impact of this technique.

Acknowledgments

The authors wish to acknowledge Ryan Grant and Simon Hammond at Sandia National Laboratories for sharing their expertise and making the experiments in this paper possible. This work was partially supported by the Sandia National Laboratories Laboratory Directed Research and Development (LDRD) program. This work was also partially supported by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research program.

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

10. REFERENCES

[1] bzip2. http://bzip.org.

[2] Extremely fast compression algorithm. https://github.com/Cyan4973/lz4.

[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. 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.

[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] 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.

it is used as a synonym for duplicated pages.

[9]These results assume 4 FIT/memory device.

[7] Colin Percival. Naive differences of executable code. http://www.daemonology.net/bsdiff/, 2010.

[8] K. B. et al. Exascale computing study: Technology challenges in achieving exascale systems, Sept. 2008.

[9] 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, (SC11)*, Nov 2011.

[10] A. S. for the Top Ten Exascale Research Challenges. Top ten exascale research challenges. Technical report, United States Department of Energy, February 2014.

[11] A. Gainaru, F. Cappello, and W. Kramer. Taming of the shrew: Modeling the normal and faulty behaviour of large-scale HPC systems. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1168–1179. IEEE, 2012.

[12] A. Gainaru, F. Cappello, M. Snir, and W. Kramer. Fault prediction under the microscope: A closer look into HPC systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC12)*, page 77, 2012.

[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, October 2010.

[14] Josh Macdonald. Xdelta. http://xdelta.org, January 2013.

[15] A. Kleen. mcelog: memory error handling in user space. In *Proceedings of Linux Kongress 2010*, Nuremburg, Germany, September 2010.

[16] Lawrence Livermore National Laboratory. Co-design at Lawrence Livermore National Lab : Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). http://codesign.llnl.gov/lulesh.php.

[17] S. Levy, P. G. Bridges, K. B. Ferreira, A. P. Thompson, and C. Trott. Evaluating the feasibility of using memory content similarity to improve system resilience. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, page 7. ACM, 2013.

[18] S. Levy, K. B. Ferreira, and P. G. Bridges. Characterizing the impact of rollback avoidance at extreme-scale: A modeling approach. In *Parallel Processing (ICPP), 2014 43rd International Conference on*, pages 401–410. IEEE, 2014.

[19] S. Levy, K. B. Ferreira, P. G. Bridges, A. P. Thompson, and C. Trott. A study of the viability of exploiting memory content similarity to improve resilience to memory errors. *International Journal of High Performance Computing Applications*, 29(1):5–20, February 2015.

[20] S. Li, K. Chen, M.-Y. Hsieh, N. Muralimanohar, C. D. Kersey, J. B. Brockman, A. F. Rodrigues, and N. P. Jouppi. System implications of memory reliability in exascale computing. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC11)*, page 46. ACM, 2011.

[21] 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.

[22] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC10)*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[23] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *Reliability, IEEE Transactions on*, 51(1):63–75, 2002.

[24] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the international symposium on Code generation and optimization*, pages 243–254. IEEE Computer Society, 2005.

[25] Sandia National Laboratories. LAMMPS molecular dynamics simulator. http://lammps.sandia.gov, Apr. 10 2013.

[26] Sandia National Laboratory. Mantevo project home page. https://software.sandia.gov/mantevo, Apr. 10 2010.

[27] Sandia National Laboratory. Kitten lightweight kernel. https://software.sandia.gov/trac/kitten, March 2012.

[28] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 297–310, New York, NY, USA, 2015. ACM.

[29] L. Xia, K. Hale, and P. Dinda. ConCORD: easily exploiting memory content redundancy through the content-aware service command. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing (HPDC)*, pages 25–36. ACM, 2014.